UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Extensiones nativas para estación de control en tierra de pilotos automáticos

Jorge López-Diéguez Asensio
Tutor: David Alcázar Llano
Ponente: Francisco Saiz López

Junio 2018

Extensiones nativas para estación de control en tierra de pilotos automáticos

AUTOR: Jorge López-Diéguez Asensio
TUTOR: David Alcázar Llano
PONENTE: Francisco Saiz López

Escuela Politécnica Superior
Universidad Autónoma de Madrid
Febrero 2018

Resumen (castellano)

Este Trabajo Fin de Grado (TFG en adelante) tiene como objetivo crear un conjunto de librerías dinámicas, DLLs, que permitan a los usuarios crear extensiones nativas para el software de control en tierra de UAV Navigation (UAVN en adelante), Visionair. Este software es el encargado de planificar las misiones que realizan los pilotos automáticos fabricados por UAVN, así como de configurarlos antes de comenzar cada misión y de comandarlos durante la misma.

Dado que Visionair se desarrolla bajo la versión 4.0 del framework .NET de Microsoft, las librerías dinámicas desarrolladas en este proyecto harán uso de la misma versión del framework, con el fin de evitar incompatibilidades.

El desarrollo de este TFG pasará por varias fases para cada uno de los elementos que lo componen: diseño, desarrollo y pruebas. En la fase de diseño se abordará qué funciones debe ofrecer cada componente y cómo se implementarán dichas funciones, en el desarrollo se creará el código necesario para ofrecer las funciones del componente, siguiendo el diseño anteriormente elegido, y durante las pruebas se validará que las funciones implementadas trabajan de manera correcta y sin errores.

En este TFG se desarrollarán cuatro librerías dinámicas, cada una encargada de ofrecer funciones diferenciadas. Visionair.Utils será la librería encargada de ofrecer clases y métodos de utilidad, como clases modelando localizaciones, métodos de conversión entre unidades, métodos de cálculo útiles, etc. Visionair.Communications manejará las comunicaciones con el piloto automático, recibiendo e interpretando los mensajes enviados por el piloto automático y creando y enviando los mensajes del usuario hacia el piloto automático. Visionair.Model proporcionará un modelo de datos representando un piloto automático, para lo cual volcará la información enviada por el mismo, haciendo uso de Visionair.Communications, en una serie de clases que la almacenarán y ofrecerán al usuario. Visionair.Extensibility proporcionará una manera de incluir extensiones nativas en Visionair, permitiendo mostrar un panel y añadir una pestaña en la sección de ajustes, así como guardar la información deseada junto al resto de información guardada por Visionair.

Con este desarrollo se logrará un conjunto de librerías que permitirán a los usuarios desarrollar extensiones nativas para Visionair sin necesidad de recibir soporte por parte de UAVN.

Palabras clave

Librería, DLL, Framework, Modelo, Canal de comunicaciones, Tests Unitarios, Protocolo, Extensiones Nativas, MEF

Abstract (English)

This Bachelor Thesis' aim is to create a suit of dynamic libraries, DLLs, which allows users to create native extensions for the UAV Navigation's (UAVN from now on) Ground Control Station software, Visionair. This software is the one in charge of planning the missions that the UAVN autopilots will perform, as well as of configuring them before starting the mission and commanding them during it.

Since Visionair is being developed on the 4.0 version of Microsoft's .NET framework, the dynamic libraries developed in this Bachelor Thesis will also use this version of the framework, to avoid incompatibilities.

The development of this Bachelor Thesis will go through several phases for each one of the elements that compose it: design, development and testing. In the design phase each component's functionalities will be discussed, as well as how will they be implemented. In the development phase the necessary code to implement the component's functionality will be written, following the chosen design. In the testing phase the written code will be validated, ensuring that the implemented functionalities work correctly and with no errors.

Four dynamic libraries will be implemented in this Bachelor Thesis, each one offering different functionalities. Visionair.Utils will be the library offering utility methods and classes, such as location modeling classes, units conversion methods, helpful calculation methods, etc. Visionair.Communications will handle the communications with the autopilot, receiving and parsing the messages sent by the autopilot and creating and sending the user's messages to the autopilot. Visionair.Model will offer a data model representing an autopilot, for which it will put the data sent by it, using Visionair.Communications, in a set of classes that will store it and offer it to the user. Visionair.Extensibility will provide a way to include native extensions in Visionair, allowing the users to show a panel and to add a tab in the settings section, as well as to save any information with the rest of the information saved by Visionair.

With this development a libraries' set will be created, which will allow the user to create native extensions for Visionair, without needing UAVN support.

Keywords

Library, DLL, Framework, Model, Communication channel, Unit Testing, Protocol, Native Extensions, MEF

Agradecimientos

Quiero agradecer en este TFG el apoyo incondicional de mi familia, así como los medios que me proporcionaron para poder llegar a realizar este TFG y, así, concluir el grado.

También agradezco a UAV Navigation la oportunidad que me brindó de incorporarme al equipo de la empresa, aceptándome desde el primer día como uno más y dándome un trato inmejorable, desde que comencé hasta el día de hoy que sigo formando parte del equipo.

En especial agradecer a mi supervisor en UAVN y mi tutor en este TFG, David Alcázar Llano, por la paciencia mostrada conmigo y por las enormes cantidades de conocimiento que he adquirido gracias a él.

INDICE DE CONTENIDOS

INDICE DE FIGURAS

INDICE DE TABLAS

1 Introducción

En esta introducción explicarán las causas y objetivos de este TFG, así como la estructura que seguirá la memoria.

1.1 Motivación

Visionair es el único software de control de tierra que es capaz de controlar cualquier piloto automático fabricado por *UAVN*, por lo que es un producto clave para el buen desarrollo y éxito de la empresa.

El desarrollo de estas librerías viene motivado por la gran variedad de aplicaciones de los pilotos automáticos de *UAVN*, pudiendo ser usados en misiones consistentes desde prácticas balísticas hasta estudios agrarios.

Dado que las funcionalidades necesarias para el manejo de un target militar no son las mismas que para el manejo de un helicóptero de aviación civil, un sistema de extensiones nativas permitiría la distribución de un programa base con funcionalidades comunes a todas las aplicaciones de los pilotos automáticos, pudiendo añadir funcionalidades específicas para cada plataforma mediante extensiones.

Estas extensiones mejorarían *Visionair* tanto en rendimiento como en espacio ocupado en disco, ya que al eliminar funcionalidades de plataformas diferentes a las del cliente el software no ejecutará ni instalará dichas funcionalidades, reduciendo el uso de CPU, memoria RAM, y almacenamiento no volátil.
Figura 1-1

1.2 Objetivos

Por los motivos mencionados en el apartado anterior, el objetivo de este TFG es crear un sistema que permita la creación de forma rápida y sencilla de extensiones que se integren de manera nativa en *Visionair*.

Para alcanzar el objetivo último de este TFG se deben cumplir otros objetivos menos ambiciosos, que en conjunto llevarán a conseguir el objetivo final:

- Diseñar e implementar una interfaz para la comunicación mediante canales físicos que abstraiga al usuario de las diferencias a la hora de enviar y recibir datos entre los distintos canales.

- Crear un sistema de interpretación y creación de paquetes siguiendo el ICD de UAVN para la comunicación con los pilotos automáticos.

- Diseñar y desarrollar un modelo de datos que represente un piloto automático, conteniendo toda la información reportada por el mismo, ya sea de forma periódica o mediante mensajes de petición y respuesta.

- Implementar un sistema que vuelque automáticamente los datos recibidos del piloto automático a su modelo de datos.

- Diseñar e implementar una librería de utilidades para facilitar al usuario el desarrollo de nuevas funcionalidades.

1.3 Organización de la memoria

Con el fin de mostrar tanto el desarrollo como el resultado final de este TFG la memoria se dividirá en los siguientes capítulos:

- **Capítulo 2: Estado del arte.** En este capítulo se expondrá el estado actual de las tecnologías usadas en el desarrollo de este TFG, valorando las ventajas y desventajas de cada una con el fin de poder elegir posteriormente la que mejor se adecue al trabajo a desarrollar.
- **Capítulo 3: Estudio previo.** Este capítulo recogerá el estudio realizado de manera previa al comienzo del TFG, incluyendo la selección de las tecnologías enumeradas en el capítulo 2.
- **Capítulo 4: Diseño.** En esta sección se detallará el diseño del sistema, así como el por qué del mismo, qué llevó a cada elección dentro del diseño del sistema a desarrollar.
- **Capítulo 5: Desarrollo.** Este capítulo mostrará el proceso de desarrollo del TFG, detallando los pasos seguidos para la implementación de cada componente.
- **Capítulo 6: Pruebas.** Esta sección recogerá los resultados del TFG, realizando pruebas para asegurar la calidad del producto final.
- **Capítulo 7: Conclusiones y trabajo futuro.** En este último capítulo se recogerán las conclusiones extraídas de la realización de este TFG y se expondrán los trabajos a realizar posteriormente.

2 Estado del arte

2.1 Framework .NET

El framework .NET es un conjunto de herramientas ofrecidas por Microsoft para el desarrollo de aplicaciones multiplataforma con soporte para la interoperabilidad entre lenguajes, es decir, es posible desarrollar una aplicación utilizando distintos lenguajes dentro de la misma y que se ejecute en distintas plataformas sin recompilar para cada una de ellas.

2.1.1 Estructura

El framework .NET se apoya en diversos conceptos:

2.1.1.1 CLI

Common Language Infrastructure (CLI) especifica un código ejecutable y un entorno para la ejecución del mismo. Esta especificación permite que varios lenguajes puedan ser ejecutados en distintas plataformas sin tener que ser diseñados específicamente para las mismas.

CLI describe tipos de datos (CTS, Common Type System), la estructura de los programas (Metadata), un conjunto de reglas que todos los lenguajes dentro de CLI deben cumplir para poder interoperar entre ellos (CLS, Common Language Specification), un lenguaje común legible por humanos al que se compilan los programas (CIL, Common Intermediate Language), y un sistema de ejecución para programas compatibles con CLI (VES, Virtual Execution System), entre otras cosas.

El CIL permite que un software, escrito en un lenguaje cualquiera, interactúe con otro software escrito en un lenguaje distinto. CIL también hace posible que un software pueda ser ejecutado en diferentes plataformas sin tener que ser diseñado específicamente para las mismas, ya que el VES será el encargado de ejecutar el código en las diferentes plataformas.

El VES se encarga de compilar en tiempo de ejecución el código CIL a código máquina compatible con la plataforma en la que se está ejecutando. El VES también proporciona servicios como el control de excepciones, manejo de hilos o gestión de memoria.

Para ejecutar un programa compatible con CLI se compila el código escrito en un lenguaje de alto nivel al CIL y se ejecuta dicho código CIL en el VES.

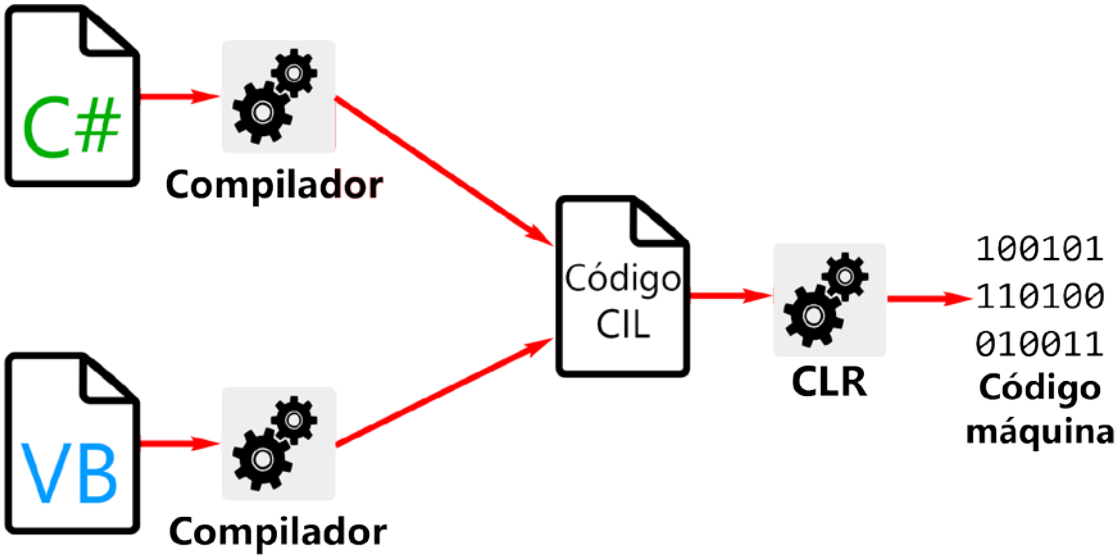

Figura 2-1: Compilación y ejecución en .NET

2.1.1.2 CLR

Common Language Runtime (CLR) es la implementación del framework .NET del VES, por lo que es el encargado de compilar *just-in-time* (JIT) el código CIL a código máquina para la plataforma en la que se ejecuta.

2.1.1.3 Ensamblados

Un ensamblado es una librería de código CIL, pudiendo ser de dos tipos: ejecutable (.exe) o librería (.dll). El código CIL contenido en los ensamblados es compilado JIT a código máquina por el VEM cuando debe ser ejecutado, lo cual hace que un mismo ensamblado pueda ser ejecutado en diferentes plataformas sin ser compilado de nuevo.

2.1.1.4 Librerías de clases

Las librerías de clases son librerías que ofrecen clases, interfaces, enumerados, y tipos de datos para ser utilizados por otros softwares. El framework .NET tiene un conjunto de librerías de clases que ofrecen una gran cantidad de funciones, facilitando en gran medida la labor del desarrollador.

Las dos librerías más importantes del framework .NET son Base Class Library (BCL), la cual ofrece funcionalidades básicas, y Framework Class Library, la cual contiene todas las funcionalidades ofrecidas en .NET.

El framework .NET utiliza las librerías de clases antes mencionadas para ofrecer la misma funcionalidad a los distintos lenguajes que soporta.

2.1.2 Lenguajes de programación

El framework .NET ofrece tres lenguajes de programación: C#, Visual Basic y F#. Todos ellos ofrecen funcionalidades muy similares, por estar enmarcadas dentro del framework .NET, por lo que no se dedicó mucho tiempo a investigar en esta línea.

C# es un lenguaje de programación orientado a objetos, cuya sintaxis es muy similar a C, C++ y Java, por lo que es sencillo aprenderlo y manejarlo si alguna vez se ha usado alguno de los lenguajes mencionados. Visual Basic es también un lenguaje orientado a objetos, con una sintaxis propia. F# es un lenguaje multiparadigma que permite emplear programación imperativa y funcional conjuntamente. F# también soporta la programación orientada a objetos, lo que lo convierte en un lenguaje muy completo.

2.2 Entorno de desarrollo

Dado que este TFG se debe desarrollar dentro del framework .NET de Microsoft, los IDEs entre los que elegir no son muy numerosos. A continuación se exponen los IDEs más interesantes evaluados para el desarrollo del proyecto.

2.2.1 Visual Studio

Visual Studio es el IDE para desarrollar en .NET proporcionado por Microsoft. Incluye soporte para el desarrollo en Visual Basic .NET y C#, así como integración con Git.

Visual Studio ofrece una gran variedad de herramientas de depuración, como acciones del recolector de basura, pila de llamadas y gestor de hilos, ente otros. Además, Visual Studio incorpora soporte para ejecutar y depurar pruebas unitarias.

Este IDE también ofrece herramientas de análisis de rendimiento. Una herramienta de análisis de rendimiento en vivo que muestra el uso de memoria y CPU del software y una herramienta de análisis más completa y detallada que recoge los datos a lo largo de una ejecución completa del software para mostrarlos posteriormente.

Visual Studio solo está disponible para Windows y MacOS, pudiendo elegir entre varias versiones gratuitas o de pago.

2.2.2 Rider

JetBrains ofrece un IDE para .NET llamado Rider. Este editor, al igual que Visual Studio soporta el desarrollo en C# y Visual Basic .NET y ofrece integración con Git.

Rider también ofrece una gran variedad de herramientas de depuración, como un gestor de hilos y una pila de llamadas muy completa y detallada, así como soporte para pruebas unitarias. Sin embargo, no ofrece un diseñador gráfico para interfaces ni una herramienta de análisis de rendimiento.

Este IDE está desarrollado sobre la misma base que Android Studio, por lo que, si se ha desarrollado alguna vez en dicho IDE, Rider será familiar y facilitará la tarea de acostumbrarse a trabajar con él y sacarle el máximo partido. Además, permite configurar el

mismo mapa de teclas y atajos que Visual Studio, facilitando la transición para estos usuarios también.

Rider es un IDE muy potente, por lo que consume una gran cantidad de recursos, sobre todo memoria RAM.

Rider es un software de pago disponible para Windows, MacOS y Linux.

2.2.3 SharpDevelop

SharDevelop es un IDE para .NET desarrollado por IC#Code, gratuito y de código abierto. Incluye soporte para Visual Basic .NET y C#, así como para Git y Subversion.

SharpDevelop incorpora varias herramientas de depuración como pila de llamadas o gestor de hilos, pero menos que Visual Studio. Se trata de un IDE ligero y rápido a la hora de iniciarse y comenzar a depurar, el cual apenas consume recursos.

SharpDevelop, a diferencia de Rider, ofrece una herramienta de análisis de rendimiento. Esta herramienta es muy completa mostrando tiempos de CPU para las llamadas a cada función, clasificados por hilos, pero no ofrece un análisis del uso de memoria. Además, a diferencia de la herramienta de análisis de rendimiento de Visual Studio, no ofrece un análisis en vivo, sireano que elabora un informe tras la finalización de la ejecución.

Durante la evaluación se descubrió que SharpDevelop no reconoce algunas expresiones de programación que si son reconocidas por Visual Studio y Rider:

- Propiedades sin cuerpo en los miembros get o set.

```
public int Value { get; set; }
```

- Ejecución de un método de un objeto condicionalmente si el objeto no es null.

```
myObject?.MyMethod(argument1, argument2);
```

- Directiva *using static*.

```
using static MyNamespace.MyClass;
```

SharpDevelop es software libre escrito en C# y ofrece un instalador para Windows.

2.3 Control de versiones

Dado que el trabajo realizado en este TFG será utilizado y evolucionado tras la conclusión de éste, se hizo necesaria la utilización de un sistema de control de versiones, tanto para conservar el estado del trabajo a cada paso del desarrollo y permitir los cambios en paralelo dentro del proyecto, como para tener una copia de seguridad en caso de que los datos almacenados en el disco duro se corrompiesen. Los sistemas de control de versiones se pueden dividir en dos categorías: centralizados y descentralizados.

2.3.1 Control de versiones centralizado

Los sistemas de control de versiones centralizados tienen un repositorio principal ubicado en un servidor. Cada usuario se conecta al servidor para obtener los archivos sobre los que desea trabajar o conocer los cambios realizados en alguno de ellos.

En los sistemas centralizados es necesaria una conexión a internet para realizar cualquier operación sobre el repositorio, permitiendo únicamente modificar los archivos locales cuando no hay disponible una conexión a internet. Esta característica también provoca que la ejecución de comandos pueda ser lenta y propensa a fallos si la conexión a internet no es rápida y fiable.

Estos sistemas se pueden administrar de manera muy sencilla, ya que solo existe una copia centralizada del repositorio. Sin embargo, si no se implementan medidas de seguridad adicionales son muy vulnerables a fallos, ya que si falla la copia principal puede resultar difícil recuperar o reconstruir el repositorio y restaurarlo al estado más reciente.

2.3.1.1 SVN

Subversion (SVN) es uno de los softwares de control de versiones más utilizado, por ser software libre y debido a su eficacia y temprano desarrollo. Se trata de un sistema de control de versiones centralizado desarrollado por CollabNet en sus inicios y por Apache en la actualidad.

2.3.2 Control de versiones descentralizado

En los sistemas de control de versiones descentralizados cada usuario tiene una copia del repositorio en su disco duro. Cuando un usuario quiere acceder a un repositorio lo clona del repositorio principal a su disco duro, obteniendo todos los archivos y su historial de cambios.

En este tipo de sistemas, dado que cada usuario tiene una copia local del repositorio, solo es necesaria una conexión a internet para clonar el repositorio, persistir cambios y descargarlos, cualquier otra operación, como volver a versiones anteriores o cambiar de rama de desarrollo, puede ser realizada sin conexión a internet gracias a la copia local del repositorio. Por otro lado, al tener cada usuario una copia local del repositorio, estos sistemas consumen más almacenamiento que con los sistemas de control de versiones centralizados. Gracias a no necesitar conexión a internet para ejecutar la mayoría de los comandos, los sistemas descentralizados son más rápidos a la hora de ejecutarlos, ya que no tienen que comunicarse con el servidor a cada comando ejecutado.

Una de las ventajas de los sistemas descentralizados es que, en caso de que se pierda información en el repositorio principal, este puede ser fácilmente restaurado utilizando las distintas copias de los usuarios.

2.3.2.1 Mercurial

Mercurial es un sistema de control de versiones descentralizado que busca ser rápido, escalable y robusto. Este sistema se empezó a desarrollar en abril de 2005 después de que BitKeeper, otro sistema de control de versiones, dejase de ofrecerse de manera gratuita.

2.3.2.2 Git

Linus Torvalds, creador de Linux, comenzó el desarrollo de Git en abril de 2005, motivado por las mismas razones que el comienzo del desarrollo de Mercurial. Los principales objetivos de Git son la rapidez, la flexibilidad, la integridad de los datos y el soporte de desarrollos paralelos.

3 Estudio Previo

En este capítulo se hablará del análisis que se llevó a cabo antes de comenzar el desarrollo del TFG, para evaluar las posibilidades y tratar de elegir la más adecuada.

3.1 Selección de herramientas

A continuación se detallan las herramientas seleccionadas para el desarrollo del proyecto, así como las razones de su elección.

3.1.1 Lenguaje de programación

El lenguaje elegido para el desarrollo del proyecto fue C#, debido a que es un lenguaje familiar por sus similitudes con otros lenguajes, muy popular, y por tanto con mucha literatura, y potente. Además, en UAVN C# es el lenguaje elegido para los nuevos desarrollos dentro de la empresa, lo que hizo que C# cobrase aún mayor importancia.

F# fue descartado debido a que es un lenguaje poco conocido y con poca literatura que debería ser aprendido por todo aquel que desarrollase o mantuviese posteriormente este proyecto. La combinación de estos dos aspectos haría que el proyecto se desarrollase lentamente y pudiese llegar a ser difícil de mantener.

Visual Basic es similar a C#, pero con una sintaxis nueva y menos familiar que la de C#. Por la gran similitud de Visual Basic y C# y los motivos expuestos anteriormente, se descartó Visual Basic y se decidió que C# sería el lenguaje en el que se desarrollaría el proyecto.

3.1.2 IDE

Tras evaluar los diversos IDEs mencionados en secciones anteriores, se decidió utilizar Visual Studio durante el desarrollo del TFG.

SharpDevelop fue el primero en ser descartado, ya que contiene menos funcionalidades que la versión gratuita de Visual Studio y su velocidad no es muy superior a la de éste.

Rider es un IDE muy potente, aunque carece de funcionalidades muy útiles como las herramientas de análisis de rendimiento. Esto, su gran consumo de recursos y su coste hicieron que fuese descartado frente a Visual Studio, ya que no posee ninguna característica importante de la que Visual Studio carezca.

3.1.3 Sistema de control de versiones

En esta sección Mercurial fue el primer sistema en ser descargado, debido a que este proyecto se convertirá en un producto de UAVN, empresa que ya tiene repositorios de SVN y Git en sus servidores. Por ello no sería práctico utilizar un sistema como Mercurial, teniendo en cuenta que Git es similar y ya está instalado en los servidores de la empresa.

Los servidores de UAVN donde se encuentran los sistemas de control de versiones solo pueden ser accedidos desde las oficinas de la empresa, por motivos de seguridad y control.

Por este motivo Git tiene ventajas sobre SVN, ya que se puede realizar cualquier operación sin conexión al servidor, excepto sincronizar el repositorio local con el repositorio maestro. Además, UAVN tiene contratados los servicios de la empresa GitLab, la cual ofrece distintas funcionalidades como un sistema de tareas, tablas para dichas tareas y diagramas de quemado, entre otras.

Por las razones expuestas se decidió emplear Git y descartar SVN como sistema de control de versiones.

3.2 Comunicación con pilotos automáticos

Para el cumplimiento de los objetivos de este TFG se hace necesaria una manera de comunicarse con los pilotos automáticos fabricados por UAVN. Estos pilotos automáticos envían y reciben paquetes usando un protocolo basado en TSIP (Trimble Standard Interface Protocol)[1], descrito a continuación:

- Existen dos caracteres especiales: DLE(0x10) y ETX(0x03).
- Los paquetes comienzan con DLE y terminan con la combinación de un DLE y un ETX, en ese orden.
- Cada paquete tiene dos bytes de identificación ID1 e ID2. ID1 no puede ser DLE.
- Todos los paquetes tienen un CRC de 32 bits al final de la carga útil del paquete.
- Todos los DLEs encontrados desde el ID2 hasta el último byte del CRC deben ser escapados con otro DLE.
- Se utiliza un orden de bytes little-endian.
- Si el paquete es enviado por un piloto automático, el paquete contiene el número de serie del mismo (entero sin signo de 16 bits) al final de la carga útil, antes del CRC.

A continuación se muestran los procesos de creación tanto de paquetes standard como de paquetes enviados por un piloto automático:

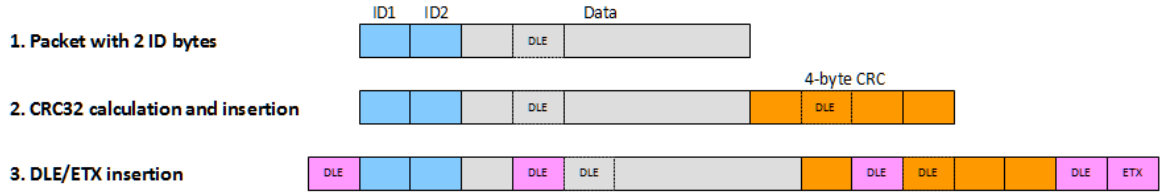

Figura 3-2: Creación de un paquete standard

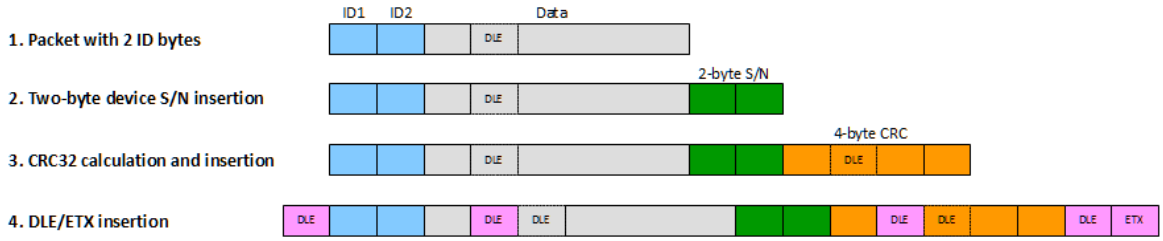

Figura 3-1: Creación de un paquete de piloto automático

UAVN ya disponía de una librería que se encarga de la creación e interpretación de los paquetes de acuerdo con el protocolo, LibUtils, además de funciones para la serialización y

deserialización de datos y cálculo de CRCs. Dicha librería se usa en los pilotos automáticos de UAVN y cuenta con pruebas unitarias, por lo que podemos afirmar que funciona correctamente. Por ello, lo ideal sería crear una DLL usando el código fuente de LibUtils. Sin embargo, LibUtils no estaba preparada para ser usada desde un proyecto de .NET, por lo que se estudiaron distintas posibilidades para crear una DLL y hacer posible su inclusión en dichos proyectos.

3.2.1 C++/CLI

C++/CLI es un lenguaje que adapta C++ a la especificación CLI (Common Language Infrastructure) de Microsoft, de la cual el framework .NET es una implementación. Es decir, es una adaptación de C++ a .NET que permite la ejecución tanto de código manejado como de código nativo, como LibUtils.

3.2.1.1 Peculiaridades de C++/CLI

Cuando código nativo y código manejado interactúan en C++/CLI, hay que tener en cuenta varios aspectos:

- En código manejado los objetos y arrays son gestionados por el recolector de basura, el cual los mueve o libera para tratar de optimizar el rendimiento del software. En código no manejado esto no ocurre, por lo que no existen mecanismos nativos automáticos para lidiar con ello.
- Si en código manejado se quieren utilizar estructuras o enumeraciones definidas en código nativo, éstas deben ser redefinidas en código manejado.

El hecho de que el recolector de basura pueda mover o liberar objetos y arrays implica que si un array u objeto es movido o liberado mientras el código nativo realiza operaciones sobre él, el resultado no será correcto, pudiendo llegar a generar violaciones de acceso. Sin embargo, existen dos soluciones a este problema, dependiendo de las necesidades del software: pin_ptr y GCHandle.

3.2.1.1.1 pin_ptr (pinning pointer)

pin_ptr[2] es un tipo de puntero que fija la dirección de memoria a la que está apuntando, es decir, evita que el recolector de basura mueva o libere el objeto o array contenido en dicha dirección. Fijar cualquier elemento de un array o cualquier miembro de un objeto fija todo el array u objeto respectivamente.

pin_ptr solo puede ser una variable local, y tan pronto como la variable deje de apuntar al objeto o array este deja de estar fijado. Esto tiene como consecuencia que cuando el programa abandone la función en la que se declaró la variable pin_ptr el objeto o array dejará de estar fijado y el recolector de basura podrá moverlo o liberarlo de nuevo.

LibUtils está diseñado para ser capaz de funcionar usando únicamente memoria estática, debido a que está integrado en el software de los pilotos automáticos de UAVN y un fallo en la reserva de memoria o violación de acceso mientras el piloto automático está volando puede ser catastrófico. Por esta razón, LibUtils guarda una referencia a un array de bytes suministrado por el usuario para usarlo como buffer para la recepción de datos.

Por lo anteriormente expuesto, pin_ptr no es una solución para este caso, ya que se podría fijar el array en la llamada a la función de inicialización de la DLL, pero tan pronto como el programa abandonase la función el array podría ser movido o liberado, dejando al código de LibUtils con una referencia incorrecta. Sin embargo, pin_ptr es la solución más adecuada para fijar los arrays utilizados en las funciones de serialización y deserialización de datos y las funciones de cálculo de CRCs, ya que solo deben ser fijados mientras se realizan las operaciones en código nativo.

pin_ptr también es útil para convertir objetos y arrays manejados en punteros nativos de C++.

3.2.1.1.2 GCHandle

GCHandle[3] es una estructura que permite evitar que el recolector de basura mueva o elimine un objeto o array manejado. A diferencia de pin_ptr, GCHandle puede ser una variable estática o miembro de clase y no liberará el objeto o array que ha fijado hasta que no se indique explícitamente, por lo que habrá que poner atención para evitar fugas de memoria.

Para fijar un objeto o array se debe realizar la siguiente llamada a la función GCHandle::Alloc (C++/CLI):

```
GCHandle handle = GCHandle::Alloc(buffer, GCHandleType::Pinned);
```

El segundo argumento de la llamada indica que el puntero no puede ser movido ni liberado. También se pueden usar el resto de los valores de la enumeración GCHandleType, pero Pinned es el valor que mejor encaja con las necesidades de LibUtils.

A la hora de liberar un objeto para que pueda ser recolectado por el recolector de basura se debe realizar la siguiente llamada a GCHandle::Free (C++/CLI):

```
tsipHandle.Free();
```

GCHandle parece ser la solución idónea para solucionar el problema planteado por LibUtils que pin_ptr no es capaz de resolver: fijar el array del que LibUtils guarda una referencia.

3.2.1.2 DLL sobre librería estática

La primera opción planteada fue la creación de una DLL encapsulando la librería estática ya existente. Para ello se crearían clases que encapsulasen las funcionalidades ofrecidas por LibUtils y las ofreciesen a través de una nueva interfaz, enlazando la librería estática durante la compilación. Sin embargo, de esta manera se introduce una capa intermedia que actúa como nexo entre el software final y LibUtils, provocando que la ejecución se ralentice.

3.2.1.3 DLL a partir del código fuente

Dado que C++/CLI admite código C++ y el código de LibUtils está escrito en este último lenguaje, se podría desarrollar una DLL que incluyese el código nativo de LibUtils, eliminando la necesidad de hacer llamadas a una librería estática y, por tanto, mejorando el rendimiento.

3.2.2 P/Invoke

Platform Invocation Services o P/Invoke es una característica del framework .NET que permite realizar llamadas a código nativo desde código manejado. En el caso de LibUtils se podría crear una DLL que únicamente actuase como pasarela entre el código manejado y código nativo exponiendo las funciones de LibUtils necesarias. Para ello se debe añadir el atributo dllexport a las funciones que se desea exponer de la siguiente manera:

```
__declspec(dllexport) void MiFuncion(int arg1);
```

De esta manera, esta DLL únicamente modificaría el código fuente de LibUtils, añadiendo el atributo dllexport a las funciones que se deseasen exponer, haciendo posible llamarlo desde código manejado importándolo de la siguiente manera:

```
[DllImport("MiDll.dll", CallingConvention = CallingConvention.Cdecl)]
public static extern void MiFuncion(int arg1);
```

Una vez importada la función, las llamadas se hacen como a cualquier otra función.

Al importar una función usando P/Invoke se debe indicar en qué DLL se encuentra la función y qué convención de llamadas usar, para que el servicio sepa cómo traducir los datos usado en código manejado a los datos usados en código nativo y viceversa. Adicionalmente, se puede indicar como traducir argumentos individuales o retornos de funciones, pero en el caso de LibUtils no sería necesario.

Con esta solución, dado que las llamadas son directas al código nativo, el usuario debería encargarse de evitar que los objetos y arrays sean movidos o liberados. Sin embargo, P/Invoke fija automáticamente los objetos y arrays pasados por argumento cuando el programa entra en una función de código nativo. De esta manera, realizar una llamada a través de P/Invoke tendría un efecto similar al del uso de pin_ptr, por lo que aparece el mismo problema ya expuesto anteriormente: se debe fijar el array que LibUtils utiliza como buffer para la recepción de datos. Esto quedaría a cargo del usuario, haciendo el uso de la librería incómodo y peligroso, ya que es fácil olvidar que se debe hacer una llamada a GCHandle.Alloc y otra llamada posteriormente a GCHandle.Free.

3.2.3 Pruebas y resultados

Para poder evaluar cada una de las opciones expuestas y elegir la más adecuada se implementaron las tres opciones. Se implementaron pequeños programas que procesaba un log grabado en un vuelo realizado por un piloto automático de UAVN, conteniendo todos los paquetes enviados a través del canal de comunicaciones.

Para que las pruebas fuesen lo más fiables posible se midieron los tiempos que cada programa tardaba en ejecutar secciones de código idénticas:

```
stopwatch.Start();
while ((bytesRead = stream.Read(logBuffer, 0, 1024)) > 0)
{
    TsipProcessRawRx(ref _tsip, logBuffer, (uint)bytesRead);
    while (TsipPendingPacketsRx(ref _tsip) > 0)
    {
        packetLength = TsipReadPacketRx(ref _tsip, packetBuffer);
    }
}
stopwatch.Stop();
```

El log utilizado para las pruebas contenía un vuelo de 15 horas, ocupando un total de 88MB. Se ejecutó cada prueba 5 veces y se obtuvo una media del tiempo empleado en procesar el archivo.

C++/CLI sobre librería estática	C++/CLI a partir del código fuente	P/Invoke
7,34096 segundos	2,68365 segundos	4,01109 segundos

Tabla 3-1: Comparación de DLLs encapsulando LibUtils

Con estos resultados, podemos apreciar que la primera solución es la más lenta de las tres con una diferencia de más de tres segundos con la siguiente más lenta. Por ello, y por la necesidad de tener tanto la librería estática como la DLL para poder funcionar, se descartó esta opción para generar una DLL encapsulando LibUtils.

Para elegir entre la opción que emplea P/Invoke y la que utiliza C++/CLI con código fuente, se tuvieron en cuenta sobre todo dos aspectos: la comodidad de uso y la facilidad de implementación e integración con el proyecto actual de LibUtils.

La opción de C++/CLI es la que mayor facilidad de uso tiene, ya que lo único que se debe hacer es llamar a las funciones que se necesiten, mientras que con la opción de P/Invoke el usuario tiene que encargarse de fijar y liberar el buffer usado por LibUtils para la recepción de datos, con los peligros que conlleva no hacer cualquiera de las dos, así como de importar las funciones que vaya a utilizar.

En cuanto a la facilidad de implementación e integración, P/Invoke sale ganando, ya que únicamente se debe añadir la directiva dllexport a las funciones que se quieran exportar, aunque se deba modificar el código de LibUtils. En el caso de C++/CLI no es necesario modificar el código de LibUtils, pero se deben crear clases que hagan las llamadas necesarias al código nativo, fijando los objetos y arrays necesarios previamente.

Finalmente se optó por implementar la solución de C++/CLI, ya que el objetivo final de este TFG es ofrecer al usuario herramientas eficientes y fáciles de utilizar, aspectos en los que la solución de C++/CLI es la mejor de las soluciones estudiadas.

4 Diseño

En esta sección se detallará el diseño del sistema a desarrollar en este TFG.

4.1 Estructura general

El título del proyecto desarrollado en este TFG será Visionair SDK, ya que permitirá a los usuarios desarrollar nuevas funcionalidades sobre Visionair. Estará compuesto por cuatro DLLs aportando distintas funcionalidades cada una.

Visionair.Communications será la librería encargada de gestionar las comunicaciones, Visionair.Model contendrá los modelos de datos, Visionair.Utils contendrá métodos y clases útiles para los desarrolladores y Visionair.Extensibility contendrá las clases que permitirán incluir controles personalizados en Visionair.

4.2 LibUtils

Las funciones contenidas en LibUtils serán necesarias para que Visionair.Communications pueda procesar los datos crudos que le lleguen. Por ello se crearán varias clases que encapsulen el uso de las funciones de LibUtils, encargandose de gestionar la identificación e interpretación de paquetes TSIP válidos, serializar y deserializar tipos de datos primitivos y calcular CRCs. Todas estas clases llamarán a los métodos de LibUtils originales, añadiendo poco o ningún procesamiento adicional, a excepción del fijado de punteros y arrays explicado en la sección de estudio previo.

4.3 Visionair.Communications

Visionair.Communications tendrá dos componentes principales: canales de comunicación e intérprete de telemetría.

4.3.1 Canales de comunicación

En Visionair SDK todos los canales de comunicaciones deben tener un API idéntico, abstrayendo al usuario de los detalles de implementación, ya sean puertos serie, comunicaciones TCP o comunicaciones UDP.

Para lograr lo anteriormente expuesto se definirá una interfaz que deberán implementar las clases que manejan los distintos canales de comunicación. Dicha interfaz se compondrá únicamente de cuatro métodos:

- Open: Se encargará de abrir el canal e inicializarlo dejándolo preparado para enviar y recibir datos.

```
void Open();
```

- Close: Cerrará el canal de comunicaciones, provocando que no se puedan leer ni escribir datos en el.

```
void Close();
```

- Read: Leerá los datos disponibles en el buffer de recepción del canal de comunicaciones. Recibirá tres argumentos: un array donde insertar los datos, el número de bytes a leer y el índice del array en el que empezar a insertar los datos.

```
int Read(byte[] buffer, int count, int offset);
```

- Write: Escribirá datos en el canal de comunicaciones. Recibirá tres argumentos: un buffer conteniendo los bytes a escribir, el número de bytes que escribir y el índice del buffer en el que empezar a coger datos para escribirlos.

```
int Write(byte[] buffer, int count, int offset);
```

Las clases modelando canales de comunicación implementarán las capas OSI de enlace de datos, red, transporte y sesión.

4.3.1.1 Comunicación serie

Para poder adaptar la comunicación por puerto serie ofrecida por el framework .NET a la interfaz diseñada para Visionair SDK se deben tener en cuenta varios aspectos.

4.3.1.1.1 Configuración del puerto serie

Para abrir un puerto serie e iniciar las comunicaciones a través del mismo es necesario indicar varios parámetros de configuración específicos, los cuales se indicarán en el constructor, para que después de instanciar la clase sea posible abstraerse de la implementación bajo la interfaz.

4.3.1.1.1.1 Nombre del puerto

Para identificar un puerto serie en el framework .NET se emplea su nombre, por lo que es necesario especificarlo para poder iniciar las comunicaciones.

4.3.1.1.1.2 Tasa de bits

La tasa de bits es el número de bits por segundo que se transmiten a través del puerto serie. Si los dos extremos de la comunicación serie no tienen configurada la misma tasa de bits, no será posible realizar una comunicación correcta.

4.3.1.1.1.3 Paridad

La paridad, si se usa, en un puerto serie puede ser par o impar. Cuando la paridad es par, se añade un bit extra a cada unidad de datos, poniendo su valor a 1 ó 0 de tal manera que el número de bits con el valor 1 en la unidad de datos sea par. Si la paridad es impar, el valor del bit de paridad será el necesario para que el número de bits con el valor 1 en la unidad de datos sea impar.

Este bit de paridad se utiliza para detectar fallos en la transmisión de datos, ya que si el número de bits con el valor 1 no coincide con lo indicado en el bit de paridad significará

que ha habido algún fallo en la transmisión. Sin embargo, este sistema es vulnerable si el número de bits corruptos es par.

4.3.1.1.1.4 Bits de datos

Los bits de datos indica el número de bits que componen una unidad de datos, pudiendo ser entre 5 y 9. Normalmente se utilizan 8 bits de datos, ya que coincide con el número de bits en un byte en la mayoría de sistemas.

4.3.1.1.1.5 Bits de parada

Los bits de parada, junto con el bit de comienzo, se utiliza para diferenciar unidades de datos correctas. El bit de comienzo siempre tiene el valor 0, mientras que los bits de parada, que pueden ser 0, 1, 1.5 ó 2, siempre tienen el valor 1.

De esta manera, el receptor de los datos puede identificar las secuencias de bits que componen una unidad de datos. Por ejemplo, si se utiliza un bit de parada, un bit de paridad con paridad par y 8 bits de datos, los datos se interpretarían como se muestra a continuación:

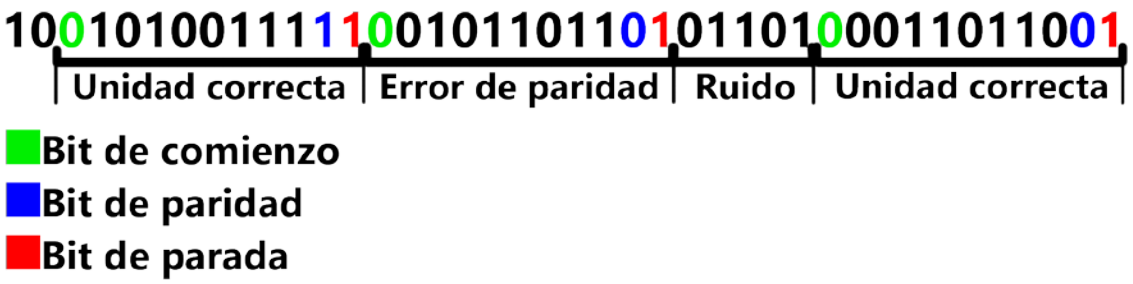

Figura 4-1: Interpretación de datos en comunicación serie

Como se aprecia en la figura, el hecho de añadir bit de paridad y bits de parada provoca que para transmitir la misma cantidad de datos útiles se deban transmitir más bits, pero añade seguridad frente a errores.

4.3.1.1.2 Recepción de datos

A diferencia de la comunicación por red, la comunicación serie no admite el concepto de paquete, por lo que la comunicación serie se compone de un flujo de bits sin más estructura que la expuesta en el apartado anterior. Por ello, cada vez que se lee de un puerto serie en el framework .NET se leen los datos que hay disponibles en el buffer de recepción, sin esperar a la llegada de un paquete como se hace en la comunicación por red.

Para unificar el comportamiento de todos los canales de comunicación se implementará un sistema para leer datos de puerto serie de la misma forma que en la comunicación por red, es decir, si no hay datos disponibles en el buffer de recepción, el hilo que realizó la llamada al método de leer quedará bloqueado hasta que llegue algún dato.

4.3.1.1.3 Envío de datos

En el caso de la escritura, aunque no exista el concepto de paquete en la comunicación serie, el envío de datos es igual que en la comunicación por red, por lo que en este caso no será necesario unificar comportamientos.

4.3.1.2 Comunicación por red UDP

El protocolo UDP no necesita ningún tipo de inicialización por no ser orientado a conexión, por lo que para empezar a transmitir y recibir únicamente es necesario configurar a qué dirección enviar y desde que puerto transmitir.

4.3.1.2.1 Configuración

La comunicación UDP se realiza a través de un socket, el cual es necesario configurar antes poder enviar y recibir datos.

Para configurar un socket es necesario indicar a qué dirección se deben enviar los datos, pudiendo ser enviados a una dirección específica o en broadcast, además de un puerto de destino. Adicionalmente, se puede seleccionar el puerto en el que recibir los datos.

4.3.1.2.2 Recepción de datos

Para recibir datos de un socket UDP únicamente es necesario leer los paquetes disponibles en el buffer de recepción, para lo cual se llama a una función de lectura que devolverá el paquete más antiguo recibido. Si no hubiera paquetes disponibles, el hilo que realiza la llamada se quedaría bloqueado hasta la recepción de un paquete.

Dado que el comportamiento buscado en la interfaz es el descrito anteriormente, no es necesario realizar modificaciones en este tipo de conexión.

4.3.1.2.3 Envío de datos

Para enviar datos a través de la conexión únicamente es necesario escribir los datos en el socket, por lo que tampoco se deben realizar modificaciones en este aspecto.

4.3.1.3 Comunicación por red TCP

El protocolo TCP es orientado a conexión, por lo que antes de empezar a transmitir y recibir datos es necesario establecer una conexión a una dirección y puerto.

4.3.1.3.1 Configuración

Al igual que en UDP, la comunicación TCP se realiza a través de un socket, al que se debe indicar a qué dirección y puerto se quiere conectar. El framework .NET ofrece una función que se encarga de establecer la conexión automáticamente, por lo que no deberemos desarrollar ninguna funcionalidad relacionada con ello.

4.3.1.3.2 Recepción de datos

Una vez establecida la conexión, la recepción de datos sigue el mismo sistema que las comunicaciones UDP, por lo que no será necesario unificar comportamientos para las conexiones TCP.

4.3.1.3.3 Envío de datos

Una vez establecida la conexión, para enviar datos únicamente es necesario escribir los datos en el socket, sin procesamiento adicional, por lo que no será necesario realizar modificaciones en esta funcionalidad.

4.3.2 Intérprete de telemetría

El intérprete de telemetría será el encargado de recibir y enviar datos a través de un canal de comunicaciones. El intérprete estará dividido en intérpretes más pequeños, encargados de manejar los paquetes de cada familia de telemetría. El intérprete principal se encargará de leer y distribuir cada mensaje entre los intérpretes específicos.

El intérprete principal ofrecerá métodos para abrir y cerrar el canal de comunicaciones al que está ligado, así como métodos para enviar paquetes. La recepción de paquetes será automática y continua, lanzando un evento cada vez que se recibe un mensaje TSIP correctamente formado, independientemente de sus contenidos.

Los intérpretes específicos se encargarán de interpretar cada paquete de su familia, lanzando un evento específico del paquete recibido con sus contenidos. Además, ofrecerán métodos para la creación de paquetes de su familia.

El intérprete de telemetría es el que implementa la capa OSI de presentación.

4.4 Visionair.Model

Visionair.Model ofrecerá una clase modelando una misión, incluyendo canales de comunicación, pilotos automáticos y estación de tierra.

Todas las clases de Visionair.Model implementarán la interfaz INotifyPropertyChanged y lanzarán el evento definido en la misma cada vez que cambie alguna de sus propiedades. Esto hará que cualquier clase pueda ser usada en un binding de XAML sin escribir código adicional.

4.4.1 Misión

El modelo de misión contendrá una lista con los canales de comunicación que el usuario desee abrir. Todo canal de comunicaciones añadido a la lista será abierto y se empezará a recibir datos del mismo. El modelo de misión se suscribirá a todos los eventos de recepción de los intérpretes de telemetría específicos, con el fin de actualizar los datos de los modelos de piloto automático y detectar pilotos automáticos que no hayan sido añadidos a la misión pero estén enviando mensajes. Cuando un nuevo piloto automático sea detectado se lanzará un evento para notificar a cualquier elemento interesado.

El modelo de misión también tendrá una lista de los pilotos automáticos, en la que se incluirán los pilotos conectados y los pilotos que el usuario quiera añadir, aunque aún no se

hayan conectado. Adicionalmente, el modelo de misión contará con un modelo de una estación de tierra.

4.4.2 Piloto automático

El modelo de piloto automático contendrá toda la información reportada por el piloto y se almacenará en una estructura de clases basada en la existente en el software de piloto automático, incluyendo clases de estimador, sensores, guiado, superficies, tiempo y servos.

4.4.3 Estación de tierra

El modelo de estación de tierra contendrá la información relativa a la posición, orientación, voltaje, velocidad de movimiento y dirección de movimiento de una estación de tierra.

4.5 Visionair.Utils

Visionair.Utils se encarga de albergar clases útiles para los desarrolladores. En esta DLL se incluirán clases modelando puntos en dos y tres dimensiones, enumeraciones usadas en el SDK que puedan ser necesarias para el usuario, clases modelando argumentos de eventos, clases con métodos de extensión, mecanismos para la conversión de unidades y otros elementos útiles para el desarrollo de extensiones.

4.5.1 Localizaciones

Para modelar localizaciones en dos y tres dimensiones se crearán dos clases: Point y Position. Point modelará puntos en dos dimensiones, guardando una latitud y una longitud, mientras que Position, que heredará de Point, guardará adicionalmente una altura. De esta manera se ofrecen mecanismos para representar puntos en el mundo tanto en dos como en tres dimensiones.

4.5.2 Enumeraciones

Las enumeraciones que serán ofrecidas al usuario son los modos del piloto automático y los tipos de software del piloto automático. La enumeración de modos contiene los distintos modos en que pueden volar los pilotos automáticos de UAVN. La enumeración de tipos de software permitirá al usuario conocer si el piloto conectado tiene software de helicóptero, ala fija o cualquier otro tipo desarrollado por UAVN.

4.5.3 Argumentos de eventos

Visionair.Utils ofrecerá clases modelando argumentos de eventos generales, como la recepción de un paquete, recepción de un ACK o recepción de un paquete conteniendo un índice. Estas clases serán útiles para poder ser heredadas por otras clases, y ser usadas en métodos para tratar argumentos de eventos de forma genérica.

4.5.4 Métodos de extensión

Durante el desarrollo de Visionair, se detectó que enviar una serie de mensajes al piloto automático y esperar sus respuestas escuchando eventos era una operación que se repetía

numerosas veces a lo largo del código. Por ello resulta útil crear un mecanismo que encapsule toda la complejidad que ello conlleva: ejecución de una acción, suscripción a eventos temporal, limitación de tiempo y reintentos.

Para escuchar las respuestas de un piloto automático es necesario suscribirse a los eventos del módulo de comunicaciones que indiquen que dicha respuesta se ha recibido. Por lo que se crearán métodos que generen IObservables que realicen una acción y esperen a que un objeto lance un evento.

La interfaz IObservable, junto a la interfaz IObserver, proporciona un sistema de notificación en el que el IObservable llama a los métodos OnNext, OnError u OnCompleted del IObserver cuando éste se suscribe. De esta manera, los métodos mencionados crearán IObservables a los que el usuario se suscribirá, iniciando la acción del IObservable y recibiendo llamadas a los métodos OnNext, OnError y OnCompleted dependiendo de si se lanza el evento esperado en un cierto límite de tiempo. Además, como en las misiones reales las comunicaciones no son siempre fiables, se implementará también un sistema de reintentos que vuelva a ejecutar la acción si no se recibe el evento en el límite de tiempo.

Se desarrollarán métodos para escuchar eventos que tengan como argumentos clases que hereden de la clase EventArgs, de la clase modelando argumentos de eventos de ACK mencionada en el punto 4.4.3 o de la clase modelando argumentos de eventos conteniendo un índice mencionado en la sección 4.4.3. En estos dos últimos casos es necesario comprobar que el evento lanzado corresponde al esperado, por lo que antes de llamar a los métodos OnNext y OnCompleted se hace la comprobación.

Hecho esto, es posible crear métodos que ejecuten los métodos descritos para una secuencia de elementos, ejecutándolos de manera ordenada y esperando a que el evento especificado sea lanzado antes de iniciar la ejecución para el siguiente elemento del IEnumerable. Para ello se crearán métodos de extensión para la interfaz IEnumerable en los que se recorra dicho IEnumerable ejecutando uno de los métodos descritos anteriormente. De nuevo, se usará la interfaz IObservable para devolver un objeto al que el usuario pueda suscribirse y comenzar la ejecución cuando desee.

Adicionalmente, se creará un método que realice la misma función que los descritos en el párrafo anterior, pero ejecutando las acciones sin esperar a que el evento sea lanzado para ejecutar la acción para cada elemento. Esto obliga a que la información del evento contenga algún elemento que identifique a qué acción hace referencia, por lo que se decidió que los argumentos del evento deberán heredar de la clase mencionada en el apartado 4.4.3 que contiene un índice.

4.5.5 Conversión de unidades

Para la conversión de unidades se creará un gestor de unidades que permita realizar conversiones entre tipos de unidades predefinidas. Este gestor además guardará las unidades seleccionadas por el usuario, permitiendo mostrar los datos al usuario en las unidades que éste seleccione de forma sencilla.

Además de las funcionalidades ya descritas, el gestor de unidades permitirá obtener los nombres y abreviaturas de las unidades predefinidas.

4.5.6 Notificación de cambio de propiedades

Para hacer el desarrollo más cómodo se creará una clase que implemente la interfaz INotifyPropertyChanged y permita lanzar el evento definido por la misma mediante una llamada a un método. El valor que añade esta clase es que obtiene el nombre de la propiedad a partir de una expresión lambda que la invoca. Esto provoca que el código sea más fácil de mantener y navegar, ya que el renombrar una propiedad con herramientas automáticas también cambiará la cadena de texto que se adjunta al evento, y al buscar los usos de la propiedad también será visible el lanzamiento del evento.

4.5.7 Temporizadores precisos

Durante el desarrollo de una misión puede ser necesario ejecutar tareas en momentos muy concretos, por lo que en Visionair.Utils se incluirá un temporizador con mayor precisión que la ofrecida por los temporizadores de .NET. Para ello se hará uso de los temporizadores nativos de Windows destinados a aplicaciones multimedia, los cuales ofrecen una mayor precisión.

4.5.8 Guardado de información global

Para el desarrollo de Visionair y sus extensiones resulta útil tener una clase donde todos los componentes de la aplicación guarden y consulten información. Por ello se creará una clase siguiendo el patrón Singleton donde se guarde información usando claves. De esta manera cualquier componente puede acceder a la información guardada en dicha clase por él mismo o por otro componente de Visionair.

4.6 Visionair.Extensibility

Visionair.Extensibility será la DLL responsable de ofrecer un mecanismo para incluir extensiones dentro de Visionair, incluyendo su interfaz gráfica y secciones de ajustes.

4.6.1 Interfaz gráfica

Las extensiones en Visionair se mostrarán como paneles sobre el mapa, permitiendo mostrarlos y ocultarlos y moverlos a una de las cuatro esquinas del mapa o a una ventana flotante.

Dado que Visionair está desarrollado en Windows Forms, Visionair.Extensibility ofrecerá un control de Windows Forms del que deberán heredar las extensiones y añadir su contenido. La elección de UAVN para nuevos desarrollos es WPF, por lo que para las extensiones desarrolladas dentro de la empresa se hará uso del control ElementHost, que permite incluir controles WPF en Windows Forms.

Este control ofrecerá funcionalidades básicas como ocultar la extensión, cambiarla de posición y sacarla a una ventana flotante. También ofrecerá un método para obtener la posición en píxeles dentro del mapa correspondiente a unas coordenadas.

Además de las funcionalidades básicas, el control tendrá diversos métodos para ser sobrescritos que serán llamados por Visionair para ofrecer las siguientes funciones: pintar en el mapa, pasar eventos de ratón, pasar presionado de teclas, guardar y cargar datos guardados junto a la información que guarda Visionair y añadir opciones al menú contextual del botón derecho del ratón.

4.6.2 Secciones de ajustes

En el caso de las secciones de ajustes, Visionair.Extensibility ofrecerá un interfaz, la cual se compondrá de métodos para actualizar y aplicar los ajustes, para guardar y cargar información junto a la información almacenada por Visionair, un control de WPF y una propiedad booleana para indicar si los ajustes han cambiado respecto a los que están aplicados, lo cual servirá para que Visionair sepa si debe aplicar o no los ajustes de la sección.

4.6.3 Importación de extensiones y secciones de ajustes

El framework .NET ofrece un sistema para descubrir y utilizar extensiones de manera sencilla: Managed Extensibility Framework (MEF)[4].

MEF permite tanto ofrecer (exportar) como utilizar (importar) componentes o extensiones en tiempo de ejecución. De esta manera Visionair podrá cargar las extensiones y secciones de ajustes automáticamente en tiempo de ejecución.

Para exportar un componente se debe hacer uso del atributo ExportAttribute, el cual puede ser heredado para crear exportaciones personalizadas añadiendo cualquier dato extra que se quiera incluir como metadatos. Para importar un elemento se debe hacer uso del atributo Import, o ImportMany si se quieren importar varios elementos. Las importaciones y exportaciones se realizan sobre un tipo, el cual se usa para relacionar unas con otras, es decir, importar únicamente los elementos que hayan hecho una exportación sobre el mismo tipo.

Para poder importar elementos se debe indicar a MEF dónde debe buscar las exportaciones, para lo cual se utilizan catálogos. Estos catálogos hacen que los componentes exportados estén disponibles para su importación y pueden ser de varios tipos como de aplicación, ensamblado o directorio. Una vez se tienen los catálogos MEF rellena las importaciones con los elementos exportados automáticamente.

En Visionair.Extensibility se definirán dos exportaciones personalizadas, una para extensiones y otra para secciones de ajustes. La exportación para extensiones incluirá el título y la posición inicial del panel de la extensión, así como en qué menú debe aparecer la opción para mostrar, ocultar y mover el panel. En el caso de la exportación de secciones de ajustes únicamente se incluirá el título de la sección como metadatos.

5 Desarrollo

En esta sección se detallará el proceso de desarrollo de Visionair SDK, aportando detalles de implementación y mostrando fragmentos de código.

5.1 LibUtils

LibUtils tiene tres funcionalidades claras definidas: identificación e interpretación de paquetes TSIP, serialización y deserialización de tipos de datos básicos y cálculo de CRCs.

Para desarrollar la nueva DLL de LibUtils se creó una solución en Visual Studio, en la que se añadieron todos los archivos fuentes de LibUtils. Se crearon archivos adicionales conteniendo clases de código manejado, las cuales se encargan de realizar las llamadas necesarias a las funciones originales de LibUtils.

5.1.1 TSIP

Para la identificación e interpretación de paquetes TSIP se creó la clase TsipParser, la cual unifica todas las funciones necesarias para el manejo del protocolo TSIP bajo una misma clase. Esta clase contiene una estructura TsipRx_t, la cual contiene todos los elementos necesarios para el funcionamiento de la máquina de estados que interpreta el protocolo TSIP, y un objeto GCHandle que fijará el buffer interno de la estructura TsipRx_t.

En el constructor únicamente se reserva memoria para la estructura TsipRx_t de LibUtils y en el finalizador se libera dicha memoria, además de liberar el buffer de la estructura para que el recolector de basura pueda volver a moverlo o destruirlo.

El método Init de TsipParser inicializa la clase y la deja lista para empezar a procesar datos llamando a la función TsipInitRx de LibUtils, donde se guarda una referencia a un buffer dado por el usuario, el número máximo de paquetes y el tamaño máximo de un paquete.

```
TsipStatus Init(array<uint8_t>^ buffer, uint32_t maxPacketLength,
                uint32_t maxPacketsToHold)
{
    _tsipHandle = GCHandle::Alloc(buffer, GCHandleType::Pinned);
    pin_ptr<uint8_t> pinnedBuf = &buffer[0];
    uint8_t * buf = pinnedBuf;
    return (TsipStatus)TsipInitRx(_tsip, buf, maxPacketLength, maxPacketsToHold);
}
```

La variable de clase _tsipHandle mantiene fijo el buffer proporcionado por el usuario durante toda la vida de un objeto TsipParser, para que el recolector de basura no lo mueva ni elimine y LibUtils no acceda a una posición incorrecta de memoria.

Las dos líneas siguientes transforman el array manejado en un puntero nativo de C++ para poder pasárselo a la función de LibUtils.

TsipStatus es una enumeración definida en LibUtils que indica en qué estado se encuentra la máquina de estados, indicando si ha habido algún error. Esta enumeración se redefinió en código manejado manteniendo los valores para poder ser usado desde código manejado.

Un aspecto importante sobre los argumentos del método Init es que el tamaño del buffer debe venir dado por los argumentos maxPacketLength y maxPacketsToHold. Para obtener el tamaño que debe tener un buffer para almacenar un máximo de x paquetes de una longitud máxima y, se debe llamar al método estático de TsipParser GetRequieredBufferSize con x e y como argumento. Este método devolverá el número de bytes que debe tener el buffer que necesita la máquina de estados para almacenar un máximo de x paquetes de longitud máxima y.

Una vez se ha llamado al método Init, la máquina de estados está lista para empezar a procesar datos. El método ProcessRaw llama a la función TsipProcessRawRx de LibUtils, la cual procesa un buffer en busca de paquetes TSIP. Si al final del buffer comienza un paquete pero no termina, la máquina de estados guarda el estado hasta la próxima llamada a TsipProcessRawRx, donde continua desde el estado en que estuviese. De esta manera, si un paquete está dividido entre dos buffers la máquina de estados será capaz de reconocerlo.

La clase TsipParser ofrece una propiedad PendingPackets, la cual encapsula una llamada a la función TsipPendingPacketsRx de LibUtils, la cual indica si se ha encontrado algún paquete y está disponible para ser leído.

Para leer un paquete se debe llamar al método ReadPacket de TsipParser. Este método llama a la función TsipReadPacketRx, la cual inserta el paquete TSIP más antiguo en un buffer proporcionado por el usuario.

LibUtils, a medida que procesa paquetes, va generando unas estadísticas sobre los datos. Estas estadísticas se pueden obtener llamando al método GetStatistics, el cual instancia la clase TsipStatistics, definida en código manejado, a partir del miembro statistics de la estructura TsipRx_t de LibUtils. Las estadísticas incluyen número de bytes y paquetes correctos e incorrectos recibidos, número de paquetes con CRC mal calculado, número de paquetes cortados, número de paquetes demasiado cortos, número de paquetes demasiado largos y número de paquetes que han sido sobrescritos por desbordamiento del buffer de recepción.

5.1.2 Serialización y deserialización

Para ofrecer las funciones de serialización y deserialización de LibUtils se creó una clase Serializa, la cual contiene métodos estáticos para transformar tipos de datos básicos a arrays de bytes y viceversa, manteniendo el orden del sistema o invirtiéndolo (little endian o big endian).

Todos los métodos se componen de dos líneas:

```
static void FloatToBytes(float value, array<uint8_t>^ buffer, uint32_t index)
{
    pin_ptr<uint8_t> buf = &buffer[0];
    floatToBytes(value, &(buf[index]));
}

static float BytesToFloat(array<uint8_t>^ buffer, uint32_t index)
{
    pin_ptr<uint8_t> buf = &buffer[0];
    return bytesToFloat(&(buf[index]));
}
```

En la primera línea se fija el buffer sobre el que se va a trabajar hasta la salida de la función, y en la segunda se llama a la función original de LibUtils. Todos los métodos trabajan con un buffer, ya sea para coger los datos o para insertarlos, por lo que en todos los casos es necesario fijarlo.

5.1.3 Cálculo de CRCs

LibUtils contiene dos funciones para realizar cálculos de CRCs, las cuales se exponen como métodos estáticos de la clase Crc32Calculator. El primer método calcula el CRC32 de un buffer y el segundo calcula el CRC32 de un buffer tomando como partida un CRC32 dado por el usuario.

Al igual que ocurre en Serialize, en estos métodos solo se realizan dos operaciones: fijar el buffer para que el recolector de basura no lo mueva ni elimine y llamar a la función de LibUtils.

```
static uint32_t CalculateCrc32(array<uint8_t>^ buffer, uint32_t start, uint32_t end)
{
    pin_ptr<uint8_t> pinnedBuf = &buffer[0];
    uint8_t * buf = pinnedBuf;
    return crc32(pinnedBuf, start, end);
}
```

5.2 Visionair.Communications

El desarrollo de Visionair SDK comenzó por el componente de más bajo nivel del mismo: las comunicaciones físicas.

5.2.1 Canales de comunicación

Como se explica en el apartado de diseño, todos los canales de comunicación implementan una interfaz que ofrece funciones básicas para la comunicación: ICommunicationChannel.

5.2.1.1 Comunicación serie

La clase encargada de manejar comunicaciones serie es ComPort. ComPort recibe en su constructor todos los parámetros necesarios para configuración de un puerto serie, siendo todos ellos opcionales excepto el nombre del puerto. Para los parámetros opcionales se utilizarán por defecto los parámetros de configuración que emplean los pilotos automáticos de UAVN:

```
public ComPort(string portName, int baudrate = 115200, Parity parity = Parity.None,
               int dataBits = 8, StopBits stopBits = StopBits.One)
{
    …
}
```

En el constructor no se realiza más acción que la inicialización de variables, sin abrir el puerto. Esta tarea se realiza en el método Open de la clase, donde se instancia la clase SerialPort, contenida en el framework .NET, con los parámetros de configuración proporcionados en el constructor. Una vez instanciada la clase se llama al método Open de SerialPort para abrir el puerto y el objeto ComPort se suscribe al evento DataReceived del

objeto SerialPort. Esta suscripción resulta necesaria, ya que, como se expuso en el diseño, se pretende unificar el comportamiento de todos los canales de comunicación ofrecidos, y los puertos serie no tienen el mismo comportamiento que los sockets, ya sean TCP o UDP, por lo que se implementó la solución explicada en el siguiente párrafo.

Dado que las llamadas a la función Read de SerialPort no son bloqueantes, se implementó un sistema en el que el hilo que llamase al método Read de ComPort quedase bloqueado si no hay datos disponibles para ser leídos. Cada vez que se recibe información en un puerto serie la clase SerialPort lanza el event DataReceived notificándolo. Cada vez que este evento es lanzado la clase ComPort lee toda la información disponible en el puerto, la mete en un array e inserta este último en una BlockingCollection. Esta clase actúa como una cola, con la peculiaridad de que al intentar extraer un elemento, si la cola está vacía, el hilo que intenta extraerlo queda bloqueado hasta que se añada algún elemento.

Cuando se llama a la función Read de ComPort se intenta extraer un elemento de la BlockingCollection, quedando el hilo bloqueado si no existe ninguno. De esta manera obtenemos el comportamiento buscado: si se intenta leer del canal de comunicaciones y no hay datos disponibles el hilo queda bloqueado hasta que haya algún dato que leer.

Cada vez que se llama a la función Read de ComPort se extrae un elemento de la BlockinCollection y se inserta en buffer proporcionado en el offset indicado. Si el elemento extraído supera la longitud máxima indicada, se guardan los datos que no caben en el buffer para ser devueltos en la siguiente llamada a Read. De la misma manera, si en la siguiente llamada se vuelve a superar el límite, se vuelven a guardar los datos para ser devueltos en la siguiente llamada, hasta que no queden datos pendientes por devolver.

La clase BlockingCollection es thread-safe, es decir, puede ser usada concurrentemente desde varios hilos sin tomar precauciones adicionales, por lo que no será necesario preocuparse por las inserciones y extracciones realizadas concurrentemente en las llamadas a Read y Write y cuando el evento DataReceived es lanzado.

El método Write no merece ninguna mención especial, ya que únicamente llama a la función Write del objeto SerialPort con los argumentos que ha recibido.

En el método Close de ComPort, además de cerrarse el puerto serie llamando a los métodos Close y Dispose de SerialPort, se vacía la BlockingCollection y se borra la suscripción al evento DataReceived de SerialPort (antes de llamar a Dispose).

5.2.1.2 Comunicación por red UDP

En el caso de las comunicaciones UDP, que serán gestionadas por la clase UdpSocket, no hubo que hacer grandes modificaciones. En el constructor se guardan la dirección a la que se enviarán y de la que se recibirán datos, así como el puerto local en el que escuchar.

De nuevo, la operación de abrir el canal de comunicaciones se realiza en el método Open. En él se instancia la clase Socket, indicando que se utilizará UDP, se llama al método Bind de Socket con el puerto local especificado en el constructor para comenzar a escuchar paquetes recibidos y se llama a Connect. Dado que UDP no es orientado a conexión, lo único que hace Connect es establecer la dirección y puerto del que se recibirán y al que se enviarán datos, descartando los paquetes que se reciban de cualquier otro host.

El método Read de UdpSocket no hace más que llamar al método Receive de Socket con los parámetros recibidos. Receive es bloqueante, por lo que no fue necesario realizar modificaciones como se hizo para las comunicaciones serie.

En el case de Write, UdpSocket únicamente llama al método Send de Socket con los argumentos recibidos.

El método Close de UdpSocket llama a los métodos Close y Dispose de Socket.

5.2.1.3 Comunicación por red TCP

Para manejar las comunicaciones TCP se creó la clase TcpSocket. Esta clase es muy similar a la clase UdpSocket, ya que también utiliza un objeto Socket para comunicarse, con la diferencia de que en el método Open únicamente se llama al método Connect de Socket. Dado que TCP es orientado a conexión, el método Connect establecerá una conexión a la dirección y puerto proporcionados desde un puerto aleatorio, por lo que no será necesario hacer una llamada al método Bind. Este comportamiento limita el uso de la clase TcpSocket a clientes, ya que el método Connect requiere que haya un servidor escuchando en la dirección y puertos seleccionados, pero, dado que los pilotos automáticos de UAVN actúan como servidor, esta funcionalidad es la necesaria.

5.2.2 Intérprete de telemetría

Como se expone en la sección de diseño, el intérprete de telemetría se compone de varios intérpretes que procesan los mensajes de familias de telemetría específica. La clase que contiene el intérprete de telemetría se llama TsipManager y se encarga tanto de recibir y procesar paquetes como de crearlos y enviarlos.

TsipManager abre un canal de comunicaciones y comienza a leer los datos que llegan por el mismo. A partir de dichos datos debe ser capaz de reconocer y procesar paquetes TSIP, por lo que se añadió un objeto TsipParser como miembro de la clase.

TsipManager recibe tres argumentos en su constructor:

```
public TsipManager(ICommunicationChannel communicationChannel,
                   uint maximumPacketLength = DEFAULT_MAX_PACKET_SIZE,
                   uint maximumPacketsToHold = DEFAULT_MAX_PACKETS_TO_HOLD)
{
    …
}
```

El primer argumento es el canal de comunicaciones que manejará el objeto TsipManager, es decir, el canal por el que enviará y recibirá datos. El segundo argumento define el tamaño máximo que podrá tener un paquete TSIP. Este argumento es opcional y por defecto vale 256. El último argumento indica el máximo número de paquetes que puede mantener la máquina de estados TSIP en su buffer de recepción antes de empezar a sobrescribir paquetes. También es opcional y por defecto vale 100.

En el constructor se guardan los valores recibidos por argumento en variables de clase, se instancia la clase TsipParser, creando el buffer que necesitará de acuerdo con los

argumentos recibidos, pero sin inicializarla, y se instancian los intérpretes de telemetría específicos.

El método Connect de TsipManager inicializa el objeto TsipParser, con el buffer y los argumentos guardados en el constructor, abre el canal de comunicaciones y comienza a leer y procesar datos del canal de comunicaciones. Esta lectura y procesamiento se realiza dentro del método privado ReceivePackets, llamado desde un hilo secundario para evitar bloquear al hilo que llama al método Connect. Para ello, dentro de Connect se realiza la siguiente llamada:

```
ThreadPool.QueueUserWorkItem(ReceivePackets, _cancellationTokenSource.Token);
```

Esta llamada encola la llamada a ReceivePackets para que se ejecute en un hilo secundario cuando alguno se encuentre disponible. El segundo argumento pasado a la función sirve para poder notificar al hilo ejecutando el método ReceivePackets que TsipManager quiere dejar de procesar datos, por ejemplo, al llamar al método Disconnect.

El método Disconnect se encarga de cerrar el canal de comunicaciones y de notificar al hilo ejecutando el método ReceivePackets que debe dejar de leer y procesar datos.

ReceivePackets es un método en el que se realizan lecturas de datos del canal de comunicaciones hasta que TsipManager indique que no se deben procesar más datos. Tras cada lectura se le pasan los datos leídos al objeto TsipParser y, cuando se detecta que hay un paquete TSIP listo para ser leído, se obtiene el paquete y se comienza a procesar. El primer paso en el procesamiento de un paquete es lanzar el evento PacketReceived, el cual notifica que se ha recibido un paquete TSIP correcto. Una vez hecho esto, se comprueba cuál es el ID1 del paquete, para identificar a que familia de telemetría pertenece y delegar la interpretación al intérprete específico adecuado.

En el destructor se cierra el canal de comunicaciones y se llama al finalizador del objeto TsipParser (llamando al método Dispose).

TsipManager cuenta con dos métodos para enviar información a través del canal de comunicaciones: SendPacket y SendRaw. SendPacket recibe un buffer y un tamaño por argumentos, creando un paquete TSIP a partir del buffer a través del método estático CreatePacket de la clase TsipParser. Una vez creado el paquete se comprueba si su tamaño es válido, en cuyo caso se envía por el canal de comunicaciones llamando al método Write del mismo. En el método SendRaw se envía un buffer recibido por argumento sin realizar ningún tipo de comprobación.

```
public void SendPacket(byte[] buffer, int count)
{
    byte[] tsipBuffer = new byte[_maxPacketLength];
    uint tsipPacketSize = TsipParser.CreatePacket(tsipBuffer, buffer, (uint)count);

    if (tsipPacketSize < MINIMUM_TSIP_PACKET_SIZE)
    {
        throw new MessageTooShortException();
    }
    else if (tsipPacketSize > _maxPacketLength)
    {
        throw new MessageTooLongException();
    }
    _communicationChannel.Write(tsipBuffer, (int)tsipPacketSize);
}
```

5.2.2.1 Intérpretes específicos

Todos los intérpretes específicos siguen la misma estructura: Tienen un método ParsePacket, el cual identifica si es un paquete de la familia y llama al método que lo procesa. Este método extrae los datos contenidos en el paquete y lanza el evento que indica que ese paquete específico ha sido recibido. Existe un método de proceso y un evento para cada paquete de la familia de telemetría. También existen métodos estáticos para crear los paquetes de la familia de telemetría que el usuario puede querer enviar al piloto automático. Adicionalmente, existe un evento genérico que indica que se ha recibido un paquete de la familia de telemetría.

Para cada evento se definió una clase que contuviera todos los datos recibidos en el paquete, además del número de serie del piloto automático que lo envió.

De esta manera, cuando TsipManager identifica a qué familia de telemetría pertenece un paquete, solo debe llamar al método ParsePacket del intérprete específico correspondiente, el cual se encargará del procesado del paquete.

5.3 Visionair.Model

El proyecto Visionair.Model contiene en un modelo la información acerca de los elementos que intervienen en las misiones realizadas por los pilotos automáticos de UAVN.

5.3.1 Mission

Toda la información contenida en Visionair.Model puede ser accedida desde la clase Mission, modelo de una misión. En ella se encuentran los objetos TsipManager con sus canales de comunicación, los pilotos automáticos y la GCS.

El constructor de la clase Mission únicamente crea las colecciones de objetos TsipManager y Autopilot (modelo para pilotos automáticos). Mission contiene métodos para añadir, obtener y eliminar objetos Autopilot de la colección interna, permitiendo al usuario manejar los pilotos automáticos que están incluidos en una misión. Además de pilotos automáticos, el usuario puede gestionar los canales de comunicación que forman parte de la misión.

Cuando se añade un canal de comunicación a un objeto Mission, este se suscribe a todos los eventos de paquetes específicos y llama al método Connect del canal de comunicaciones. Cada vez que se lanza un evento de paquete específico, Mission actualiza la información del objeto Autopilot con el número de serie del piloto automático que envió en mensaje. Si no existe ningún objeto Autopilot con dicho número de serie en la colección del objeto Mission, este lanzará el evento NewAutopilotDetected con el número de serie del piloto que envió el mensaje para notificar que se ha detectado un piloto que no está presente en la misión, permitiendo que el usuario pueda añadirlo o realizar cualquier otra acción.

5.3.2 Autopilot

Autopilot es la clase que contiene el modelo para pilotos automáticos. Contiene toda la información relacionada con un piloto automático, almacenada en una estructura basada en la existente en el código de los pilotos automáticos de UAVN.

En la clase Autopilot se guarda información como el número de serie, el nombre que se le quiere dar al piloto, las versiones de software, el uso de CPU, información de estimación, de sensores, de guiado, de superficies de control, de servos y de tiempos, alarmas activas en el piloto, historial de posiciones del piloto o estela (track) y referencia de velocidad (IAS o GPS) y altura (barométrica o GPS) usadas, entre otras.

5.3.3 Gcs

La clase Gcs contiene el modelo para estaciones de control en tierra. Esta clase contiene la posición en tres dimensiones de la GCS, su orientación, su velocidad y dirección de movimiento, su voltaje y su historial de posiciones o estela (track). Se trata simplemente de una clase que guarda información, sin ninguna otra funcionalidad.

5.4 Visinoair.Utils

Visionair.Utils contiene clases útiles para desarrolladores.

5.4.1 Data

Data es una clase que implementa un sistema para guardar información que puede ser accedida desde cualquier lugar si se conoce la clave asociada. Esta clase es un Singleton, por lo que todo el código tendrá una referencia del mismo objeto Data.

Data permite añadir y eliminar variables, consultar y modificar su valor, consultar si existe una variable a partir de su clave y suscribirse a los eventos de cambio de una variable. Data contiene un diccionario que asocia claves de texto a objetos de la clase DataElement.

La clase DataElement contiene un objeto object, que puede ser consultado o modificado. Cada vez que es modificado se lanza el evento DataChanged, que notifica que el valor ha cambiado. El valor antes de la modificación también puede ser consultado.

Cuando se llama al método SetValue de Data, se modifica el valor del DataElement asociado a la clave dada únicamente si el nuevo objeto y el contenido en DataElement no son iguales, evitando lanzar el evento DataChanged de manera innecesaria.

Los objetos DataElement de Data no son accesibles desde fuera del objeto, por lo que para suscribirse al evento DataChanged se debe llamar al método SubscribeToOnChangeEvent de Data indicando la clave de la variable a la que suscribirse y el manejador del evento.

```
public void SubscribeToOnChangeEvent(string key, EventHandler<DataChangedEventArgs>
eventHandler)
{
    if (!IsVariableDefined(key))
    {
        throw new VariableNotDefinedException("Variable not defined.");
    }
    _dataDictionary[key].DataChanged += eventHandler;
}
```

Para transmitir información en los eventos de cambio de valor, se creó la clase DataChangedEventArgs, la cual únicamente contiene un valor antiguo y un valor nuevo.

La clase Data permite también consultar el valor de una variable a través de un indizador que recibe una clave y devuelve el objeto asociado a dicha clave.

```
public object this[string key]
{
    get
    {
        if (!IsVariableDefined(key))
        {
            throw new VariableNotDefinedException("Variable not defined.");
        }
        return _dataDictionary[key].Value;
    }
}
```

5.4.2 PreciseTimer

Dado que los temporizadores ofrecidos por .NET no ofrecen gran precisión, se implementó un temporizador que ofreciese mayor precisión. Para ello se utilizan los temporizadores que ofrece Windows para aplicaciones multimedia. Para poder utilizar dichos temporizadores es necesario importar las funciones que los gestionan, contenidas en la librería de sistema winmm.dll:

```
private delegate void TimerEventDel(int id, int msg, IntPtr user, int dw1, int dw2);
[DllImport("winmm.dll")]
private static extern int timeBeginPeriod(int msec);
[DllImport("winmm.dll")]
private static extern int timeEndPeriod(int msec);
[DllImport("winmm.dll")]
private static extern int timeSetEvent(int delay, int resolution,
        TimerEventDel handler, IntPtr user, int eventType);
[DllImport("winmm.dll")]
private static extern int timeKillEvent(int id);
```

La función timeBeginPeriod solicita una resolución para el temporizador, indicando si es posible alcanzar dicha resolución o no. timeEndPeriod elimina la resolución previamente solicitada por timeBeginPeriod.

La función timeSetEvent inicia un temporizador con el periodo, resolución, manejador, datos de usuario y tipo de temporizador indicados. La resolución indica los milisegundos de tolerancia para lanzar el evento de completado de ciclo, el manejador es la función a la que se llamará cada vez que el temporizador complete un ciclo, los datos de usuario serán pasados al manejador cuando sea llamado y el tipo de temporizador indica si debe reiniciarse cuando complete un ciclo. Esta función devuelve el identificador del temporizador creado.

La función timeKillEvent elimina el temporizador correspondiente al identificador pasado.

La clase PreciseTimer imita el API de la clase System.Timers.Timer de .NET, siendo controlada a través de los métodos Start y Stop y la propiedad Autoreset.

El constructor de PreciseTimer recibe y guarda el periodo del temporizador en milisegundos y crea un delegado para el evento de completado de ciclo del temporizador multimedia.

La propiedad AutoReset modifica el valor del argumento eventType de la función timeSetEvent, cambiando entre temporizador periódico y no periódico.

El método Start llama a la función timeSetEvent con los valores especificados en el constructor, una resolución de cero y el método TimerCallback como manejador. El método TimerCallback lanza el evento Elapsed de PreciseTimer si el identificador del temporizador coincide con el que devolvió la función timeSetEvent.

El método Stop llama a la función timeKillEvent con el identificador del temporizador iniciado anteriormente.

5.4.3 Unidades

Visionair.Utils ofrece un sistema de conversión de unidades, manejado por la clase UnitsManager. Esta clase contiene una instancia de la clase UnitsType para cada tipo de unidad (altura, distancia, velocidad lineal, presión, coordenadas, ángulos, aceleraciones y velocidades angulares). Dicha clase contiene las unidades de un tipo seleccionadas por el usuario y las unidades disponibles de dicho tipo contenidas en objetos de la clase Unit. La clase Unit contiene el nombre y abreviatura de una unidad concreta.

La clase UnitsManager instancia la clase UnitsType para cada tipo de unidad, insertando los datos de todas las unidades de cada tipo, es decir, su nombre y abreviatura. Las unidades disponibles de cada tipo están definidas en diferentes enumeraciones.

UnitsManager ofrece los siguientes métodos para convertir valores entre dos unidades de un mismo tipo: entre dos unidades dadas, de unidades del usuario a unas unidades dadas y de unas unidades dadas a unidades del usuario.

```
public static double Convert(AltitudeUnits originUnits, double originValue,
AltitudeUnits destinationUnits)
{
    double result = originValue;
    if (originUnits != destinationUnits)
    {
        switch (originUnits)
        {
            case Enums.AltitudeUnits.Feet:
                result = originValue * FEET_TO_METERS;
                break;
            case Enums.AltitudeUnits.Meters:
                result = originValue * METERS_TO_FEET;
                break;
        }
    }
    return result;
}
```

```
public double Convert(AltitudeUnits originUnits, double originValue)
{
    return Convert(originUnits, originValue, _altitudeUnits.UserUnits);
}

public double Convert(double originValue, AltitudeUnits destinationUnits)
{
    return Convert(_altitudeUnits.UserUnits, originValue, destinationUnits);
}
```

Además de los métodos de conversión, UnitsManager permite obtener el nombre y abreviatura de cada unidad y las unidades de cada tipo seleccionadas por el usuario.

5.4.4 Posiciones

Las clases Point y Position modelan posiciones en dos y tres dimensiones respectivamente. La clase Point guarda una latitud y una longitud y la clase Position, que hereda de Point, guarda además una altura.

5.4.5 IObservable

La interfaz IObservable, junto con la interfaz IObserver, proporciona un mecanismo para enviar notificaciones. El IObserver se suscribe al IObservable y éste llama a los métodos OnNext, OnError y OnCompleted del primero cuando corresponda. Cuando un IObserver desea recibir notificaciones de un IObservable únicamente debe llamar al método Subscribe del mismo.

Este mecanismo resulta muy útil para ejecutar una acción y esperar una respuesta, operación muy frecuente en la comunicación con los pilotos automáticos de UAVN.

Para aprovechar las funcionalidades ofrecidas por estas interfaces se crearon tres métodos que crean un IObservable, al cual se puede suscribir un IObserver para ejecutar una acción y esperar una respuesta. El primer método es WaitForResponse:

```
public static IObservable<T> WaitforResponse<T>(Action request, object eventTarget,
        string eventname, int timeoutMilliseconds = 3000, int retries = 2)
        where T : EventArgs
{
    return Observable.Create<T>(observer =>
    {
        int tries = retries + 1;
        IObservable<EventPattern<T>> obs_evt = Observable.Return<Unit>(new Unit())
            .Do(aux => request())
            .Zip(Observable.FromEventPattern<T>(eventTarget, eventname),
                (a, b) => b)
            .Timeout(TimeSpan.FromMilliseconds(timeoutMilliseconds))
            .Retry(tries)
            .Take(1);
        obs_evt.Subscribe(evt => observer.OnNext(evt.EventArgs),
                          ex => observer.OnError(ex),
                          () => observer.OnCompleted());
        return new BooleanDisposable();
    });
}
```

Este método crea un Observable el cual ejecuta una acción, proporcionada a través del argumento request, cuando un IObserver llama al método subscribe. Este Observable se

une con otro, que transforma un evento de .NET en un Observable. El evento transformado viene dado por los argumentos eventTarget (objeto que lanza el evento) y eventName (nombre del evento). De esta manera, cuando el objeto proporcionado lanza el evento indicado el Observable llamará al método OnNext del IObserver que se haya suscrito.

Adicionalmente, el método Timeout añade un tiempo máximo para que el evento sea lanzado, el método Retry permite que si el evento no ha sido lanzado en el tiempo dado por Timeout, se vuelva a iniciar el Observable el número de veces pasado como argumento, y por último, el método Take provoca que únicamente se espere a un lanzamiento del evento indicado, es decir, que cuando se lance el evento una vez se llamará a los métodos OnNext y OnCompleted del IObserver suscrito.

Con toda esta lógica, el usuario puede crear un observable que mande un mensaje al piloto automático y espere a que Visionair.Communications lance el evento de recepción de la respuesta.

El método WaitForAck utiliza la misma lógica que el método anterior para esperar a la recepción de un ACK de un paquete con los IDs pasados por argumento. Para ello restringe que el tipo de los argumentos del evento, el parámetro T del método, sea de tipo AckEventArgs (argumentos para el evento de recepción de un ACK). De este modo tendrá acceso a los IDs del paquete al que se está haciendo ACK y podrá filtrarlos haciendo uso del método Where:

```
Where(evt =>
{
    return ((evt.EventArgs.Id1 == familyId) && (evt.EventArgs.Id2 == packetId));
})
```

WaitForAck es exactamente igual que WaitForResponse, excepto porque el método Where se llama tal y como se ilustra en la muestra de código anterior sobre el siguiente Observable, pasado como argumento al método Zip:

```
Observable.FromEventPattern<T>(eventTarget, eventname)
```

De esta manera, si se lanza el evento de recepción de ACK para un paquete que no es el deseado, se ignorará el evento y se seguirá esperando.

El método WaitForIndex únicamente se diferencia de WaitForAck en que el parámetro T debe ser de tipo IndexEventArgs, para tener acceso al índice del paquete recibido, en lugar de recibir dos IDs recibe un índice para filtrar y que el método Where se llama con el siguiente argumento:

```
Where(evt =>
{
    return (evt.EventArgs.Index == index);
})
```

5.4.5.1 Extensiones de IEnumerable

Al igual que se hace individualmente en WaitForIndex, el usuario puede querer ejecutar una acción varias veces esperando a cada una de las respuestas. Para lograrlo se crearon tres métodos de extensión para la interfaz IEnumerable.

ToSequentialPattern devuelve un IObservable que ejecuta una acción, esperando a un evento, para cada uno de los elementos de un IEnumerable<int> de manera secuencial, es decir, espera a que se lance el evento a causa de la acción ejecutada para un elemento antes de ejecutar la acción para el siguiente. Esto se consigue haciendo uso del método WaitForResponse explicado anteriormente:

```
public static IObservable<T> ToSequentialPattern<T>(this IEnumerable<int> ie,
        Action<int> request, object eventTarget, string eventname,
        int timeoutMilliseconds = 3000, int retries = 2) where T : EventArgs
{
    return Observable.Create<T>(observer =>
    {
        IEnumerator<int> enumerator = ie.GetEnumerator();
        enumerator.MoveNext();
        IObservable<T> obs_evt = IObservableUtils.WaitforResponse<T>(
            () => request(enumerator.Current), eventTarget, eventname,
            timeoutMilliseconds, retries).Repeat().Take(ie.Count());
        obs_evt.Subscribe(evt =>
        {
            enumerator.MoveNext();
            observer.OnNext(evt);
        }, ex => observer.OnError(ex), () => observer.OnCompleted());
        return new BooleanDisposable();
    });
}
```

Este método repite el IObservable devuelto por WaitForResponse tantas veces como elementos tenga el IEnumerable sobre el que se ejecuta. A pesar de estar repitiendo el mismo IObservable, las acciones que ejecuta el mismo se ejecutan sobre distintos elementos, ya que se utiliza un enumerador en la llamada a la acción y se avanza el mismo cada vez que se lanza el evento esperado (se llama a OnNext en el IObserver).

El método ToSequentialIndexPattern solo se diferencia del método ToSequentialPattern en que se utiliza WaitForIndex en lugar de WaitForResponse y en que T debe heredar de IndexEventArgs, ya que se está utilizando el método WaitForIndex.

El método ToBatchPattern realiza las mismas funciones que ToSequentialPattern, pero no de manera secuencial, sino esperando un tiempo dado por argumento entre ejecuciones de la acción. Esta característica provoca que la lógica del método sea mucho más compleja.

Dado que los mensajes pueden llegar a la vez o fuera de orden, ToBatchPattern restringe que los argumentos del evento esperado hereden de IndexEventArgs, para poder diferenciarlos.

En primer lugar, el método crea un observable a partir del evento indicado, al cual se suscribe. Cada vez que se llama al método OnNext, se añade el índice recibido a una lista de índices recibidos.

Después crea una lista idsToSend, donde añade todos los elementos del IEnumerable sobre el que se está ejecutando el método. Una vez creada dicha lista se suscribe a un Observable que se ejecuta cada cierta cantidad de milisegundos pasada por el usuario por argumento, siempre y cuando no haya ocurrido un timeout o se haya completado el lanzamiento de todos los eventos esperados. En este Observable es donde se ejecuta la acción para cada elemento del IEnumerable. Para ello se toma el primer elemento de la lista idsToSend, se

realiza la acción para el mismo y se elimina de idsToSend. De este modo en cada iteración del IObservable se realizará la acción para un elemento diferente sin repetirse.

Por último, el método crea y se suscribe a un Observable que se ejecuta cada vez que pasa el periodo de timeout. Cada vez que esto ocurre se comprueba si se ha llegado al número máximo de timeouts o reintentos. En caso negativo, se añaden a la lista idsToSend todos aquellos índices que no se hayan recibido todavía y no estén ya contenidos en idsToSend. De esta manera el segundo Observable seguirá enviando los índices que tuviera pendientes, además de aquellos que se enviaron, pero aún no se han recibido.

En caso de que se alcanzara el máximo de timeouts, se comprobaría si falta algún índice por recibir, en cuyo caso se llamaría al método OnError del IObserver con una excepción en la que se inserta una lista con los índices no recibidos, en caso contrario se llama al método OnCompleted del IObserver.

5.4.6 PropertyChangedNotifier

PropertyChangedNotifier implementa la interfaz INotifyPropertyChanged, y su objetivo es permitir lanzar el evento PropertyChanged de manera fácil y con un mantenimiento fácil. Para ello, se creó el método OnPropertyChanged, el cual recibe una expresión lambda invocando la propiedad que ha cambiado y lanza el evento PropertyChanged con el nombre de dicha propiedad.

Para obtener el nombre de la propiedad se sirve del método estático GetPropertyName de la clase Utils.

```
public static string GetPropertyName<TProperty>(Expression<Func<TProperty>> property)
{
    var lambda = (LambdaExpression)property;
    MemberExpression memberExpression;
    if (lambda.Body is UnaryExpression)
    {
        var unaryExpression = (UnaryExpression)lambda.Body;
        memberExpression = (MemberExpression)unaryExpression.Operand;
    }
    else
    {
        memberExpression = (MemberExpression)lambda.Body;
    }
    return memberExpression.Member.Name;
}
```

Este método recibe una expresión lambda que accede a una propiedad. Para obtener la propiedad a la que accede la expresión lambda se debe obtener la MemberExpression contenida en la misma. MemberExpression es una clase que representa el acceso a un campo o propiedad, exactamente lo que contiene la expresión lambda, a partir de la cual podemos obtener el nombre del campo o propiedad al que se accede.

Haciendo uso de este método, se podría lanzar el evento PropertyChanged con el texto “MyProperty” con la siguiente llamada al método OnPropertyChanged de PropertyChangedNotifier:

```
OnPropertyChanged(() => MyProperty);
```

5.5 Visionair.Extensibility

Visionair.Extensibility es el encargado de proporcionar el mecanismo para incluir extensiones en Visionair. Para ello, tal y como se explicó en el diseño, hará uso de MEF, creando dos atributos de exportación (para extensiones y secciones de ajustes) e importando las clases que hagan uso de los mismos.

5.5.1 Widget

Widget es el control de Windows Forms del que deben heredar las interfaces gráficas de las extensiones. Este control tiene una barra en la parte superior donde se muestran el título, el botón de ocultar y el botón de mover a una ventana flotante.

Widget tiene una propiedad Position, la cual es usada por Visionair para saber si mostrar o no el control y si debe colocarlo dentro del mapa y dónde. Esta propiedad es del tipo EnumPosition, enumeración que contiene los valores Hidden (oculto), TopLeft (esquina superior izquierda), TopRight (esquina superior derecha), BottomLeft (esquina inferior izquierda), BottomRight (esquina inferior derecha), BelowFirstRow (segunda fila en la esquina superior izquierda), Full (ocupando el mapa completo) y Floating (en una ventana independiente).

Relacionada con esta propiedad esta la propiedad Tsmi, la cual contiene el ToolStripMenuItem con el que se muestra, oculta y mueve el widget de la extensión desde Vsionair. Cuando el widget está visible, la propiedad Checked de Tsmi se pone a true, mientras que cuando está oculto se pone a false. Cada vez que el usuario hace click en el ToolStripMenuItem asociado al widget, Visionair llama al método CyclePosition. Este método oculta o muestra la ventana flotante del widget cuando Position es Floating, o rota la posición del Widget en caso contrario, en el siguiente orden: TopLeft, TopRight, BottomLeft, BottomRight y Hidden. Si Position es Hidden, CyclePosition la pasa a TopLeft.

Los objetos de la clase Widget pueden elegir si desean recibir eventos del ratón, para lo cual se debe usar la propiedad ReceiveMouseEvents. Únicamente cuando la propiedad ReceiveMouseEvents tiene el valor true, Visionair llamará a los métodos MouseDown (cuando se presiona un botón del ratón), MouseUp (cuando se suelta un botón del ratón), MouseMove (cuando se mueve el ratón), MouseDoubleClick (cuando el usuario hace doble click) y GetRightClickOptions (cuando se está abriendo un menú contextual).

Los métodos KeyDown y KeyUp son llamados por Visionair cada vez que el usuario pulsa o libera una tecla, indicando la tecla pulsada o liberada. Con estos métodos y los expuestos en el párrafo anterior, las extensiones pueden obtener input del usuario fuera del Widget.

La clase Widget también permite pintar en el mapa a sobrescribiendo el método Draw. Visionair llama al método Draw de cada Widget cada vez que actualiza el mapa. Este método recibe un objeto Graphics por argumento, sobre el que deberá pintar para mostrar el contenido que desee mostrar.

Para permitir a las extensiones añadir opciones a los menús contextuales, Visionair llama al método GetRightClickOptions (indicando sobre que elemento se hizo click, la latitud y la longitud) cada vez que se vaya a mostrar un menú contextual. Con esta información la

extensión podrá añadir las opciones que desee dependiendo del elemento sobre el que se hizo click (piloto automático, posición de modo Hold, punto del plan de vuelo o mapa).

Además de los métodos ya expuestos, la clase Widget cuenta con los métodos ExportWidgetSettings e ImportWidgetSettings. Estos métodos son invocados por Visionair cuando guarda y carga información respectivamente y permite a las extensiones guardar información con el mismo mecanismo que lo hace Visionair. Para guardar información, Visionair proporcionará un objeto XElement, el cual representa un nodo XML donde insertar cualquier información. A la hora de cargar la información, el mismo nodo será pasado como argumento en el método ImportWidgetSettings.

5.5.1.1 WidgetExportAttribute

Para permitir la exportación de clases que heredan de Widget desde DLLs externas se definió el atributo WidgetExportAttribute. Esta clase hereda de ExportAttribute e implementa la interfaz IWidgetMetadata. Dicha interfaz únicamente contiene tres propiedades, InitialPosition, Menu y Title, y obliga que al ser usado para exportar un Widget se deba indicar su posición inicial, el menú en el que aparecerá y el título que tendrá. De esta manera, cuando Visionair importe los objetos Widget, cada Widget tendrá en sus metadatos la información proporcionada a través de IWidgetMetadata.

5.5.2 ISettingsSection

En el caso de las secciones de ajustes, Visionair.Extensibility proporciona una interfaz: ISettingsSection. Esta interfaz contiene dos propiedades y cuatro métodos. La primera propiedad es SettingsView, de la clase Control de WPF, cuyo contenido será el mostrado cuando se seleccione la sección de ajustes. La otra propiedad es HaveSettingsChanged, de tipo booleano, que permitirá saber a Visionair si ha habido algún cambio en la configuración.

UpdateSettings y ApplySettings son los métodos que serán llamados cuando se deban actualizar y aplicar los ajustes respectivamente.

Para el guardado y cargado de datos se ofrecen los métodos ExportSectionSettings e ImportSectionSettings respectivamente. Estos métodos reciben los mismos argumentos que los métodos ExportWidgetSettings e ImportWidgetSettings, ya que serán llamadas del mismo modo y en los mismos momentos que éstos.

5.5.2.1 SettingsSectionExportAttribute

Al igual que para Widget se definió un atributo de exportación para las secciones de ajustes: SettingsSectionExportAttribute. De igual forma se definió una interfaz ISettingsSectionMetadata, cuya única propiedad es SectionTitle, indicando el título de la sección. SettingsSectionExport implementa la interfaz ISettingsSectionMetadata, obligando a que las clases importadas por Visionair que implementen la interfaz ISettingsSection indiquen un título.

6 Pruebas

Antes de poder dar por finalizado el proyecto se deben realizar pruebas para asegurarse de que el software desarrollado no contiene errores elementales.

Para hacer las pruebas más rápidas y automáticas, se usaron las herramientas de pruebas unitarias (Unit Testing) incluidas en el framework .NET.

Para LibUtils no se definieron pruebas nuevas, ya que el código usado ya posee pruebas unitarias creadas antes del comienzo de este proyecto.

6.1 Visionair.Communications

En primer lugar, se crearon las pruebas para los canales de comunicación, por ser el nivel más bajo en el que se basa el correcto funcionamiento de la librería. Estas pruebas unitarias validan el correcto funcionamiento de las clases que implementan la interfaz ICommunicationChannel. Para ello, se crean parejas de objetos ICommunicationChannel de la misma clase y se conectan entre sí, para enviar y recibir información comprobando que la información enviada por un canal es la misma que se recibe en el otro.

En las pruebas de comunicación serie es necesario configurar, a través de una herramienta externa, los puertos COM1 y COM2 para que se comuniquen entre sí. En el caso de TcpSocket se creo una clase auxiliar que actúa como servidor, comunicando dos sockets TCP durante el desarrollo de las pruebas. Para las pruebas de UdpSocket no fue necesario realizar ningún desarrollo o configuración extra.

Una vez se comprobó que ICommunicationChannel era fiable, se crearon las pruebas para la clase TsipManager. En estas pruebas se utilizan dos objetos de la clase UdpSocket conectados entre sí. Uno de dichos sockets se utiliza para instanciar la clase TsipManager, mientras que por el otro se enviarán los datos de prueba.

La primera prueba de TsipManager consiste en comprobar que interpreta los paquetes descritos en el ICD de manera correcta, para lo cual se mandan buffers conteniendo cada uno de los paquetes soportados creados a mano de acuerdo con el ICD. Una vez se ha enviado el paquete, se comprueba que el evento correspondiente a la recepción del mismo sea lanzado conteniendo los datos correctos.

La segunda prueba consiste en comprobar que los paquetes creados por los distintos intérpretes de telemetría son correctos, para lo cual se define a mano cada uno de los paquetes soportados de acuerdo con el ICD y se compara con el paquete creado por los intérpretes específicos.

Por último, se prueban los métodos de envío de información: SendPacket y SendRaw. Para probar SendPacket se envía un buffer con ciertos datos, comprobando que en el otro extremo se recibe dichos datos encapsulados en un paquete TSIP correcto. SendRaw se prueba enviando un buffer con datos y comprobando que en el otro extremo de la conexión se reciben esos mismos datos.

6.2 Visionair.Model

Las pruebas de Visionair.Model consisten en el envío de paquetes a través de una conexión UDP, por ser la más sencilla de usar, en la que escucha una instancia de la clase Mission, comprobando posteriormente que los datos del modelo se han actualizado de forma acorde a la información contenida en el paquete enviado. Se comprueba además que se lanza el evento PropertyChanged con las propiedades que han cambiado en los objetos correspondientes.

Se comprueba también que el evento NewAutopilotDetected se lanza únicamente cuando se inserta en el paquete un número de serie distinto al de los pilotos añadidos al modelo.

6.3 Visionair.Utils

En Visionair.Utils solo se crearon pruebas para las clases que contienen funcionalidad.

La clase PropertyChangedNotifier no necesita pruebas específicas, ya que se utiliza en el modelo y ya se está comprobando que se lanzan los eventos PropertyChanged adecuados.

Para probar la clase PreciseTimer se instancia con un periodo de 10 milisegundos, incrementando un contador cada vez que se cumple el tiempo. Usando temporizadores de .NET se recoge el valor del contador cada segundo durante 10 segundos. Una vez terminado este tiempo se comprueba que la media de las muestras recogidas es igual a 100, las veces que debería saltar PreciseTimer en un segundo con el periodo configurado.

Para probar la clase Data se obtiene la instancia y se definen varias variables, tras lo cual se añaden suscripciones a los eventos de cambio de las mismas. Una vez hecho esto se llama al método SetValue con diferentes valores, comprobando que el evento se lanza únicamente cuando el nuevo valor es distinto al que existía anteriormente. También se comprueba que tras cancelar la suscripción no se reciben más eventos de cambio.

La clase UnitsManager se prueba llamando a cada método de conversión con valores cuya conversión ya es conocida, comprobando que el valor devuelto es correcto. También se prueban los métodos de conversión dependientes de las unidades de usuario cambiando dichas unidades.

Dado que ya se ha probado el modelo y sabemos que funciona correctamente, los métodos de extensión de IEnumerable se prueban enviando un plan de vuelo a través de un canal de comunicaciones UDP, indicando que se debe esperar al evento PropertyChanged del plan de vuelo de un piloto contenido en un objeto Mission. Al usar un plan de vuelo, cuyo evento de recepción hereda de IndexEventArgs, es posible probar los métodos ToBatchPattern, ToSequentialPattern y ToSequentialIndexPattern. También se comprueba que el método OnError se llama cuando no se lanza el evento esperado en el tiempo dado, para lo cual se utiliza una acción vacía en la llamada a los métodos. Para ello se mide el tiempo desde que se llama al método de extensión hasta la llamada a OnError.

6.4 Visionair.Extensibility

Esta librería no se probará en el proyecto, ya que corresponde a Visionair probar las funcionalidades.

7 Conclusiones y trabajo futuro

7.1 Conclusiones

El conjunto de librerías desarrollado en este proyecto permitirá la integración de extensiones nativas en Visionair de manera sencilla y con mantenimiento sencillo. Para crear una extensión o sección de ajustes, se deberá crear una solución de Visual Studio, o un proyecto en cualquier otro IDE, que construya una DLL, añadir referencias a las DLLs de Visionair SDK de las que se desee hacer uso, y comenzar a desarrollar. Una vez completado el desarrollo, se pondrá la DLL generada en la ubicación que Visionair indique y automáticamente se incluirá la extensión o sección de ajustes dentro de Visionair, sin tener que realizar ninguna otra acción. Cuando se quiera actualizar Visionair SDK, únicamente se deberán sustituir las DLLs del proyecto de la extensión o sección de ajustes y solucionar los errores que se generen, únicamente si el API ha cambiado, recompilar y sustituir la DLL usada por Visionair.

Sin embargo, esta no será su única función. Visionair SDK también se convertirá en el núcleo de Visionair, siendo utilizado tanto para las comunicaciones a través de Visionair.Communications como para el almacenado de datos en Visionair.Model y la resolución de problemas frecuentes a través de Visionair.Utils. Visionair.Extensibility también permitirá añadir paneles al mapa desde dentro de Visionair, sin necesidad de generar una nueva DLL con la funcionalidad deseada.

UAVN dispondrá de una nueva fuente de ingresos a través de la venta de Visionair SDK ayudando al desarrollo y crecimiento de la empresa.

7.2 Trabajo futuro

Tras el desarrollo del proyecto, queda pendiente para el futuro su integración en Visionair, permitiendo que el desarrollo de extensiones y del propio Visionair se haga de una manera ordenada y sencilla.

UAVN también deberá valorar las opciones de venta de este producto, ya que puede resultar un producto muy útil a la hora de ofrecer la integración de nuevas funcionalidades personalizadas de los clientes dentro de Visionair, sin tener que pagar por su desarrollo directamente a UAVN.

El código desarrollado se convertirá en la base sobre la que desarrollar Visionair, por lo que su mantenimiento y actualización serán de suma importancia. Por ello se hizo hincapié en la facilidad de mantenimiento durante el diseño del proyecto.

Referencias

[1] Trimble Navigation Limited, TSIP Reference, 1-3 (sección 1.4), Abril 1999.
[2] Microsoft documentation about pin_ptr (https://msdn.microsoft.com/es-es/library/1dz8byfh.aspx)
[3] Microsoft documentation about GCHandle (https://msdn.microsoft.com/en-gb/library/system.runtime.interopservices.gchandle(v=vs.110).aspx)
[4] Microsoft MEF documentation (https://docs.microsoft.com/en-us/dotnet/framework/mef/index)

Glosario

UAVN	UAV Navigation.
Visionair	Software de control de tierra de UAVN.
ICD	Interface Control Document (Documento de Interfaz de Control). Documento describiendo la interfaz de control de un sistema.
GCS	Ground Control Station (Estación de Control en Tierra).
Framework	Conjunto de funcionalidades ofrecidas dentro de un paquete de software para facilitar el desarrollo de aplicaciones. Puede incluir compiladores, librerías, APIs, etc.
API	Application Programming Interface (Interfaz de programación de aplicaciones). Es un conjunto de funciones ofrecido por una librería para ser utilizada desde otro software,
IDE	Integrated Development Environment (Entorno de Desarrollo Integrado). Se trata de un programa que facilita el desarrollo de aplicaciones ofreciendo un editor, compilador, depurador y cualquier otra herramienta útil para el desarrollo.
Little-endian	El orden en el que se representan los bytes de un dato que ocupa más de un byte es de menos significativo a más significativo.
DLL	Librería de enlace dinámico. Son librerías que contienen código ejecutable que únicamente se carga y ejecuta bajo demanda.
CLI	Common Language Infrastructure. Es la especificación desarrollada por Microsoft que describe el código y entorno para permitir la ejecución de múltiples lenguajes en diferentes plataformas sin tener que diseñarse específicamente para cada una. El framework .NET está desarrollado de acuerdo con CLI.
Bytecode	Es un tipo de conjunto de instrucciones que es ejecutado o transformado por un intérprete. De esta manera el mismo código puede ser utilizado en distintos entornos, cambiando únicamente el intérprete.
CIL	Common Intermediate Language. También conocido como MSIL (Microsoft Intermetiate Language), es el lenguaje legible de más bajo nivel definido por CLI. Es de tipo bytecode y es usado por el framework .NET.
CLR	Common Language Runtime. Es un entorno de ejecución para el código del framework .NET. Es la implementación de Microsoft de CLI y se encarga de compilar el código CIL a código máquina nativo de la plataforma en la que se ejecuta. CLR realiza una compilación en tiempo de ejecución o JIT (Just In Time).
Código manejado	Es el código que se ejecuta bajo la gestión de CLR.
Código nativo	También conocido como código no manejado, es el código que se ejecuta fuera de CLR.
Socket	Representa un extremo en una conexión, identificándolo mediante una dirección y un puerto. Esto permite que a través de un mismo canal físico se transmitan los datos de diversas conexiones.
Binding de XAML	Mecanismo de XAML para actualizar de forma automática la información mostrada en la interfaz.

IAS	Indicated Airspeed. Velocidad del aire medida por un sensor de presión en la aeronave, sin aplicar correcciones.